kaspersky

KasperskyOS Community Edition SDK 2.0

© 2026 AO Kaspersky Lab

Contents

About this Guide

What's new

About KasperskyOS Community Edition SDK

- Distribution kit
- System requirements
- Included third-party libraries and applications
- Drivers included in the SDK
- Limitations and known issues
- Critical changes in version 2.0

Training scenarios

- Obtaining and installing the SDK
- Quick Start. Introduction to the SDK
- Working with the extension for Visual Studio Code
- Running an example on the hardware platform
- Debugging an example in Neovim

KasperskyOS operating system overview

- Overview
- Architecture and startup of a KasperskyOS-based solution
- KasperskyOS kernel subsystems
- IPC
  - IPC mechanism
  - IPC control
  - Transport code for IPC
  - IPC between a process and the kernel
- Resource Access Control

Getting started with the SDK

- The KasperskyOS Developer's Quick Start Guide
- Using a Docker container
- Installation and removal
- Configuring the development environment
- KasperskyOS SDK Extension for Visual Studio Code
  - Extension functions
  - Extension usage scenarios
    - Installing and removing the extension
    - Working with a Docker container in Visual Studio Code
    - Updating the extension
    - Creating a project in Visual Studio Code
    - Starting a basic KasperskyOS-based solution image
    - Building programs in Visual Studio Code
    - Starting programs in a basic solution image
    - Starting programs using a launch.json startup configuration in Visual Studio Code
    - Viewing event logs

- Debugging a program that is part of a KPA package
  - Connecting a debugger when starting a program in a basic image
  - Connecting a debugger while a program is running from a basic image
- Extension reference information
  - Buttons in the lower panel
  - Commands in the command palette
  - Controls in the list of devices and emulators
  - Extension settings
- Building and running examples
  - Building the examples
  - Running examples on QEMU
  - Preparing Raspberry Pi 4 B to run examples
  - Preparing OrangePi RV2 to run examples
  - Running examples on Raspberry Pi 4 B or OrangePi RV2
- Development for KasperskyOS
  - Starting processes
    - ServiceManager system program
    - Starting processes using the system program ExecutionManager
    - kl.ProductRomfsLoader initializing program and product_init.yaml file
    - Example init descriptions
  - File systems and network
    - Working with the VFS system program
      - Contents of the VFS component
      - Creating a static IPC channel to VFS
      - Building a VFS executable file
      - Including VFS functionality in a program
      - VFS startup parameters and environment variables
      - Mounting file systems when VFS starts
      - Using VFS backends to separate data streams
      - VFS relay mechanism
    - Considerations for working with file systems
      - Supported file systems
      - File system mounting functions
      - Notification subscription functions
      - Function for tracking connected drives (kos_devfs.h)
    - Considerations for working with a network
      - Configuration and status of network interfaces (kos_net.h)
      - Vector network functions sendvec() and recvvec()
    - Dynamically configuring the network stack
  - IPC and transport
    - Creating IPC channels
    - Adding an endpoint from KasperskyOS Community Edition SDK to a solution
    - Creating and using your own endpoints
      - Overview: IPC message structure
      - Example generation of transport methods and types
      - Working with an IPC message arena

- Transport code in C++
  - Dynamically creating IPC channels for C++ development using DCM
- Working with KPA packages
  - KPA package manifest
    - Application object
    - Platform object
    - List of "components" objects
    - List of runConfiguration objects
    - PrivateStorage object
  - Tools for managing KPA packages
    - cas-pack tool
    - cas-inspect tool
    - cas-pm tool
  - PackageManager component usage scenario
- Using the command line of the KasperskyOS kernel

KasperskyOS API

- Return codes
- libkos library
  - Managing handles (handle_api.h)
    - Handle permissions mask
    - Creating handles
    - Transferring handles
    - Duplicating handles
    - Dereferencing handles
    - Revoking handles
    - Closing handles
    - Getting a security ID (SID)
    - OCap usage example
  - Managing virtual memory (vmm_api.h)
    - Allocating and freeing memory
    - Creating shared memory
    - Preparing ELF image segments to be loaded into process memory
    - Receiving information about memory
  - Using a working set (ws_api.h)
  - Allocating and freeing memory (alloc.h)
  - Using DMA (dma.h)
  - Memory-mapped I/O (mmio.h)
  - Managing interrupt processing (irq.h)
  - Managing threads (high-level API thread.h)
  - Managing threads (low-level API thread_api.h)
  - Managing processes (high-level API task.h)
  - Managing processes (low-level API task_api.h)
  - Initializing IPC transport for interprocess communication and managing IPC request processing (transport-kos.h, transport-kos-dispatch.h)
  - Initializing and de-initializing IPC transport for querying the security module (transport-kos-security.h)
  - Using a thread pool to process IPC requests (dispatch_pool.h)
  - Generating random numbers (random_api.h)

Getting and changing time values (time_api.h)
Using notifications (notice_api.h)
Using synchronization primitives (event.h, mutex.h, rwlock.h, semaphore.h, condvar.h, sync_api.h, sync_types.h, sync.h)
- Using events (event.h)
- Using mutexes (mutex.h)
- Using read-write locks (rwlock.h)
- Using semaphores (semaphore.h)
- Using condition variables (condvar.h)
- Static initialization of synchronization primitives (sync_types.h)
- Using futexes (sync.h)

Managing I/O memory isolation (iommu_api.h)
Using queues (queue.h)
Using memory barriers (barriers.h)
Executing system calls (syscalls.h)
Managing IPC (ipc_api.h)
Using sessions (session.h)
Using KosObjects (objects.h)
Using KosObject containers (objcontainer.h, objcontainernolock.h, basecontainer.h)
- Using KosObjContainers (objcontainer.h)
- Using KosObjContainerNolock (objcontainernolock.h)
- Using KosBaseContainers (basecontainer.h)

Using KosObject identifiers (objhandles.h)
Using KosStrings (strings.h)
KasperskyOS kernel XHCI DbC driver management (xhcidbg_api.h)
Getting IPC handles and RIIDs to use statically created IPC channels (sl-static.h)
Power management (pm_api.h)
Getting values of performance counters (value_api.h)
Using the hardware abstraction layer (hal_api.h, hal_common.h, hal_env.h, hal_privregs.h)
Flushing the processor cache and getting processor cache info (cpucache.h)
Using the ROMFS file system (romfs.h)
Using the ROMFS file system (high-level API fs.h)
Using the ROMFS file system (low-level API fs_api.h)
Preparing loadable ELF image segments for creating a process (elf_api.h)
Hardware platform firmware communication (efi_api.h)
Converting UUID and GUID presentation formats (uuid.h, guid.h, uuid_common.h)
Asynchronous formatted output (log.h)
Multi-step thread-safe initialization and finalization (init_fini.h)
Using specialized loadable sections of an ELF image containing data object arrays (iterable_section.h)
Logging the operation of programs (trace.h)
Getting the build ID from an ELF image (elf_id.h)
Getting build IDs for ELF images loaded into process memory (task_deb_api.h)
Gaining access to named blobs (blob_api.h)
Adapting the scheduler to a hardware platform that uses processors of varying performance (sched_api.h)
Using a memory pool (mempool.h)
Using data containers (valcontainer.h)

- POSIX support
  - POSIX support limitations
  - POSIX implementation specifics
  - Concurrently using POSIX and the libkos API
- Obtaining statistical data on the system
  - Obtaining statistical data on the system through the libkos library API
  - Obtaining statistical data on the system through the libc library API
- Dynamically creating IPC channels using the DCM system program
  - Using the base API (dcm_api.h)
  - Using the auxiliary API for clients (connect.h)
  - Using the additional server API for servers (pub.h)
  - Using the additional API for clients (subscr.h)
- LogRR component
  - Adding a log server and output channels to a solution
  - Getting log entries when running in QEMU
  - Using macros to send messages to a log
  - Log levels
  - LogRR program API
    - logrr_clog library
      - ClogInit(), ClogLog() functions and CLOG macro
      - LogrrLogLevel enumerated type
    - logrr_cpp library
      - Init(), Log() methods and LOG macro
      - LogLevel class
- ExecutionManager component
  - IProcessControl interface
  - IApplicationController interface
  - IStateProvider interface
  - ISystemController interface
- PackageManager component
  - IPackageController interface
  - IPackageManifest interface
- TLS Terminator component
  - Adding TLS Terminator to a KasperskyOS-based solution
  - TLS Terminator startup parameters and environment variables
  - Extending TLS Terminator functionality
- Toybox component
  - Adding Toybox to a KasperskyOS-based solution
- Kaspersky Control Flow Monitor component

Building a KasperskyOS-based solution

- An overview of the solution building process
- Using CMake from the contents of KasperskyOS Community Edition SDK
  - CMakeLists.txt root file
  - CMakeLists.txt files for building applications
  - CMakeLists.txt file for building the solution image
  - product_init.yaml.in template

- CMake libraries in KasperskyOS Community Edition SDK
  - platform library
  - nk library
    - Creating EDL files and generating transport code in C
      - generate_edl_file ()
      - nk_build_idl_files ()
      - nk_build_cdl_files ()
      - nk_build_edl_files ()
      - Parameters for generating transport code in C
    - Generating transport code in C++
      - add_nk_idl ()
      - add_nk_cdl ()
      - add_nk_edl ()
      - bind_nk_targets ()
  - kpa library
    - add_kpa_package()
    - add_kpa_component()
  - build_sd_image library
  - minimal_kos library
    - generate_qemu_target()
    - build_rootfs()
    - build_einit()
    - build_romfs()
  - build_vendor_rom library
  - kl_install_packages library
- Using dynamic libraries
  - Life cycle of a dynamic library
  - Building dynamic libraries
- Cross compilers

Debugging programs in a KasperskyOS-based solution

- Using the GDB debugger
  - Initial steps of debugging in QEMU
  - Preparations for debugging on the hardware platform
  - Initial steps of debugging on the hardware platform
  - Working with processes and threads
- Analyzing stack backtrace data when a process abnormally terminates
- Using a handle sanitizer
- Debug adapter for the Neovim editor
  - Debugging an application
  - Adapter reference information

Developing security policies

- Formal specifications of KasperskyOS-based solution components
  - Names of process classes, components, packages and interfaces
  - EDL description
  - CDL description

IDL description
IDL data types
Integer expressions in IDL
Describing a security policy for a KasperskyOS-based solution
General information about a KasperskyOS-based solution security policy description
PSL language syntax
Setting the global parameters of a KasperskyOS-based solution security policy
Including PSL files in a KasperskyOS-based solution security policy description
Including EDL files in a KasperskyOS-based solution security policy description
Creating security model objects
Binding methods of security models to security events
Creating security audit profiles
PSL data types
Examples of binding security model methods to security events
Example descriptions of basic security policies for KasperskyOS-based solutions
Examples of security audit profiles
KasperskyOS security models
Auxiliary security models
Pred security model
Bool security model
Math security model
Struct security model
Precedence and associativity of auxiliary security model expressions
Base security model
Regex security model
HashSet security model
HashSet security model object
HashSet security model init rule
HashSet security model fini rule
HashSet security model add rule
HashSet security model remove rule
HashSet security model contains expression
StaticMap security model
StaticMap security model object
StaticMap security model init rule
StaticMap security model fini rule
StaticMap security model set rule
StaticMap security model commit rule
StaticMap security model rollback rule
StaticMap security model get expression
StaticMap security model get_uncommitted expression
StaticMap security model initialized expression
Flow security model
Flow security model object
Flow security model init rule
Flow security model fini rule
Flow security model enter rule

Flow security model allow rule
Flow security model query expression
Mic security model
Mic security model object
Mic security model create rule
Mic security model delete rule
Mic security model execute rule
Mic security model upgrade rule
Mic security model call rule
Mic security model invoke rule
Mic security model read rule
Mic security model write rule
Mic security model query_level expression
Using templates when creating a KasperskyOS-based solution security policy description
Methods of KasperskyOS core endpoints
Virtual memory endpoint
Physical memory endpoint
I/O endpoint
Threads endpoint
Handles endpoint
Processes endpoint
Synchronization endpoint
File system endpoint
Time endpoint
Hardware abstraction layer endpoint
KasperskyOS kernel XHCI DbC driver management endpoint
Audit endpoint
Profiling endpoint
I/O memory isolation management endpoint
Power management endpoint
Notifications endpoint
Trusted Execution Environment endpoint
IPC management endpoint
Hardware platform firmware communication endpoint
Statistics endpoint
BLOB endpoint
Endpoint for adapting the scheduler to a hardware platform that uses processors of varying performance
Process diagnostics endpoint
Security patterns for development under KasperskyOS
Distrustful Decomposition pattern
Secure Logger example
Separate Storage example
Defer to Kernel pattern
Defer to Kernel example
Policy Decision Point pattern
Privilege Separation pattern
Device Access example

- Information Obscurity pattern
  - Secure Login (Boost, TLS-terminator) example
- Examples included in the SDK
  - hello example
  - echo example
  - ping example
  - net_with_separate_vfs example
  - net2_with_separate_vfs example
  - embedded_vfs example
  - vfs_extfs example
  - multi_vfs_ntpd example
  - multi_vfs_dns_client example
  - multi_vfs_dhcpcd example
  - mqtt_publisher (Mosquitto) example
  - mqtt_subscriber (Mosquitto) example
  - gpio_input example
  - gpio_output example
  - gpio_interrupt example
  - gpio_echo example
  - kcfm_echo_server example
  - logrr example
  - pcre example
  - i2c_ds1307_rtc example
  - iperf_separate_vfs example
  - uart example
  - spi_check_regs example
  - barcode_scanner example
  - watchdog_system_reset example
  - shared_libs example
  - pal_tests example
  - hello_from_rust example
  - hello_corrosion example
  - mass_storage example
  - framebuffer_qr_writer example
  - dump_collector example
  - kpa_example
  - uart_toybox example
  - perfcnt example
  - vfs_relay example
  - virtiofs example
- Appendices
  - Information about certain limits set in the system
  - Preparing the WiFi driver for operation in Raspberry Pi 4 B
  - Developing drivers for KasperskyOS
- Glossary
  - Address Space Layout Randomization
  - Affinity mask

Application
Arena chunk descriptor
Arena descriptor
ASLR
Callable handle
Capability
CDL
Client
Client library of the solution component
Client process
Condition variable
Constant part of an IPC message
CPU mask
Critical section
DCM handle
Describing a security policy for a KasperskyOS-based solution
Direct memory access
DMA
DMA buffer
EDL
Empty process
Endpoint
Endpoint interface
Endpoint method
Endpoint method ID
Event
Event mask
Execute interface
Formal specification of the KasperskyOS-based solution component
Frozen state of a process
Futex
Global system information
GSI
Handle
Handle dereferencing
Handle inheritance tree
Handle permissions mask
Handle transport container
Hardware interrupt
IDL
Inferior
Init description
Interface method
Interprocess communication
IPC
IPC channel
IPC handle

IPC message

IPC message arena

IPC request

IPC response

IPC transport

KASLR

KasperskyOS

KasperskyOS security model

KasperskyOS-based solution

KasperskyOS-based solution component

Kernel Address Space Layout Randomization

KPA package

KPA package manifest

KSM

KSS

Listener handle

MDL

MDL buffer

Memory barrier

Memory Descriptor List

Message signaled interrupt

MID

MMIO

MMIO memory

Mutex

Notification receiver

OCap

PAL

PCB

Process

Process control block

Program

PSL

Read-write lock

Recursive mutex

Resource

Resource consumer

Resource integrity level

Resource provider

Resource transfer context

Resource transfer context object

RIID

Runtime Implementation Identifier

SCP

Security audit

Security audit configuration

Security audit data

Security audit profile
Security audit runtime-level
Security context
Security event
Security ID
Security interface
Security model expression
Security model method
Security model object
Security model rule
Security module decision
Security pattern
Security pattern system
Security policy for a KaspersкyOS-based solution
Security template
Seed
Semaphore
Server
Server library of the solution component
Server process
SID
SMP
Static connection page
Subject integrity level
System program
System resource
TCB
Thread
Thread control block
Thread identifier
Thread local storage
TID
TLS
Transport code
Transport library
User resource
User resource context
Working set
WS
Zero page deduplication
ZPD
Legal notice
Licensing
Data provision
Information about third-party code
Trademark notices

About this Guide

This Guide is intended for software developers who work on KasperskyOS-based software/hardware systems. The main part of this Guide provides the following information:

- KasperskyOS conceptual guides.
- Step-by-step tutorials showing you how to work with the KasperskyOS Community Edition SDK.
- How-to guides that describe the actions that a developer should take to get specific practical results. A how-to guide may be presented as an example of code with explanations.

  For example, how-to guides are provided in the following sections:

  - Quick Start Guide to developing KasperskyOS solutions
  - Getting started with the SDK, Development for KasperskyOS
  - KasperskyOS API
  - Building a KasperskyOS-based solution
  - Debugging programs in a KasperskyOS-based solution
  - Developing security policies
- Reference guides:
  - API information
  - Syntax of shell commands, GDB commands, and CMake commands
  - Syntax of specialized languages used to automatically generate source code
  - POSIX support details
  - Information on the KasperskyOS kernel methods used for a security policy description

How-to guides and reference guides may be combined into one section.

What's new

KasperskyOS Community Edition SDK 2.0 has the following new capabilities and refinements:

> Due to modifications made to SDK components, you must make changes to the application code that was developed using KasperskyOS Community Edition SDK version 1.4 before using that code with KasperskyOS Community Edition SDK version 2.0. For more details, refer to Critical changes in version 2.0.

- Added support for a new architecture: RISC-V. The OrangePi RV2 single-board computer was chosen as the hardware platform. Implemented the Instruction Set Architecture (ISA): rv64gc_zba_zbb_zbc_zbs.
  - The following drivers are implemented for the OrangePi RV2 hardware platform:
    - BSP
    - Ethernet
    - SD/emmc
    - UART
    - Pinctrl
    - GPIO
    - I2C
    - SPI
    - USB (HID + Mass storage)
- The KasperskyOS Community Edition SDK 2.0 release includes three versions of the SDK:
  - Version for working in the QEMU emulator
  - Version for the Raspberry Pi 4 Model B hardware platform
  - Version for the OrangePi RV2 hardware platform.
- A transition to the arm64 virt-qemu virtual platform and the addition of the Virtiofs virtual file system have been implemented to increase performance for the QEMU emulator.
- Expanded the list of supported drivers for the Raspberry Pi 4 Model B hardware platform. The following drivers have been added:
  - USB-UART
  - Pinctrl

- The internal architecture and the process of loading OS components and system programs have been modified. Currently, KasperskyOS Community Edition SDK includes the following:
  - InitROMFS boot image containing the minimum set of system programs required to run a solution and applications packed into KPA packages.
  - VendorROMFS image containing additional system programs and drivers required for working with the emulator or a specific hardware platform on which the solution will run.
- Modified list of examples:
  - Added the virtiofs and kcfm_echo_server examples.
  - The kcfm_hello and kcfm_http_server examples have been removed.
- Added the capability to use the debug adapter in the VIM editor in experimental mode for the QEMU emulator in the SDK.
- Updated system requirements: the Ubuntu GNU/Linux 24.04 "Noble Numbat" operating system is required for SDK installation.
- Updated Developer's Guide, including:
  - Added descriptions of improvements and new features introduced in KasperskyOS Community Edition SDK 2.0.
  - Added training scenarios.
- Updated the following third-party libraries and applications:
  - Autoconf (v.2.72)
  - Binutils (v.2.46.1)
  - GoogleTest (v.1.14.0)
  - libxml2 (v.2.15.4)
  - MbedTLS (v.3.6.6)
  - OpenSSL (v.3.5.7)
  - python (v.3.14.0)
  - rust (v.1.77.2)
  - sqlite (v.3.53.2)
  - Zlib (v.1.3.2)

KasperskyOS Community Edition SDK 1.4 has the following new capabilities and refinements:

- QEMU support has been moved to a separate SDK. The KasperskyOS Community Edition SDK 1.4 release comes with three versions of the SDK: a version for running in the QEMU emulator and separate versions for each of the supported hardware platforms. For more details, refer to Distribution kit.
- Expanded the list of supported drivers for the Raspberry Pi 4 Model B hardware platform. The following drivers have been added:
  - FrameBuffer
  - Wi-Fi
  - Camera
  - MBIM
- Expanded the list of supported drivers for the Radxa ROCK 3A hardware platform. The following drivers have been added:
  - PWM
  - PCIE
  - NVMe
  - Watchdog
  - Wi-Fi
  - FrameBuffer
  - Camera
  - MBIM
  - Bluetooth
- Added support for new devices:
  - USB camera
  - USB modem
  - SSD drive
- Added the `Dump Collector` component, which lets you generate dumps via the API of the `ExecutionManager` system program if a process abnormally terminates.
- Added the Kaspersky Control Flow Monitor (KCFM) component which is designed to monitor and control the performance of processes by monitoring the control flow.

- Added terminal support via the Toybox component, which is an implementation of the toybox command line tool adapted to work in KasperskyOS.
- Restored support for some performance counters, and added the perfcnt example to the SDK.
- Expanded the list of examples to include a demonstration of working with KPA packages.
- Excluded the `MessageBus` and `koslogger` components from the SDK.
- Updated the Developer's Guide.
- Added the following third-party libraries and applications:
  - Immediate-Mode-UI/Nuklear (v4.12.7)
  - Skia (v.m112)
  - zxing-cpp/zxing-cpp (v.2.3.0)
- Updated the following third-party libraries and applications:
  - CMake (v.3.29.0)
  - Libtool (v.2.4.7) -- v.2.4.2 was deleted
  - Binutils (v.2.44)
  - QEMU (v.9.2.4)
  - Boost (v.1.85.0)
  - fmt (v11.0.2)
  - iperf (v.3.19.1)
  - libpcap (v.1.10.5)
  - libxml2 (v.2.12.10)
  - Mbed TLS (v.3.6.4)
  - google/re2 (v.2022-04-01)
  - wpa_supplicant (v.2.11)
- Excluded the following third-party libraries and applications from the SDK:
  - civetweb
  - spdlog

KasperskyOS Community Edition SDK 1.3 has the following new capabilities and refinements:

- Added support for the Radxa ROCK 3A hardware platform.
- Added an extension for the source code editor Visual Studio Code to integrate it with KasperskyOS Community Edition SDK.
- Added the PackageManager component that lets you install KPA packages in an operational KasperskyOS-based solution, remove KPA packages and receive information about them; and added tools for managing KPA packages.
- Added the capability for debugging using the GDB debugger.
- KasperskyOS Community Edition SDK now includes tools for development in Rust and program examples.
- Added the capability to work with USB drives.
- The toolchain included in KasperskyOS Community Edition SDK now uses the Clang compiler.
- Added the LogRR component, which is a system for logging information about the operation of other programs.
- Updated Developer's Guide, including:
  - Added descriptions of improvements and new features introduced in KasperskyOS Community Edition SDK 1.3.
  - Added descriptions of scenarios for working with libkos library interfaces.
  - Added the Introduction section.
  - Added the KasperskyOS Developer's Quick Start Guide section.
- Added the following third-party libraries and applications:
  - abseil-cpp (20211102.0)
  - clang (17.0.6)
  - clang-format (13.0.1)
  - corrosion-rs/corrosion (0.2.2)
  - ftpd (2.3.0)
  - libyaml (0.2.5)
  - python (3.12.2)
  - google/re2 (2022-02-01)
  - rust (1.59)
  - wpa_supplicant (2.10)

- Updated the following third-party libraries and applications:
  - binutils
  - boost
  - json-schema-validator
  - libevdev
  - libtool
  - mbedtls
  - qemu
  - usb
- Excluded the following third-party libraries and applications from the SDK:
  - autotools-wrappers

KasperskyOS Community Edition SDK 1.2 has the following new capabilities and refinements:

- Updated system requirements: the Ubuntu GNU/Linux 22.04 "Jammy Jellyfish" operating system is required for SDK installation.
- Added capability to use dynamic libraries.
- Added capability to use a hardware watchdog on the Raspberry Pi 4 Model B.
- Added ExecutionManager component designed for creating, starting, and stopping processes.
- Added script for automatically setting environment variables used by SDK tools.
- Added data transmission to Kaspersky servers when starting a build of examples from the SDK. Data is transmitted to account for the number of users of KasperskyOS Community Edition SDK and to obtain information about the distribution and use of KasperskyOS Community Edition SDK. You can disable this functionality.
- Updated Developer's Guide, including:
  - Added section titled "Working with an IPC message arena".
  - Added section titled "Information about certain limits set in the system".
  - Added descriptions of scenarios for working with libkos library interfaces.
  - Updated instructions on building and running solution security policy tests.
  - Added glossary.

- Added the following third-party libraries and applications:
  - Guidelines Support Library (GSL) (2.1.0)
  - json_scheme_validator (2.1.0)
  - libpcap (1.10.4)
  - libunwind (1.6.2)
- Updated the following third-party libraries and applications:
  - libxml2
  - Mbedtls
  - Mosquitto
  - OpenSSL
  - spdlog
  - sqlite
  - fmt
  - zlib
  - flex
  - bison
  - QEMU
- Excluded the following third-party libraries and applications from the SDK:
  - ffmpeg
  - opencv
  - libjpeg-turbo
  - libpng
  - protobuf

KasperskyOS Community Edition SDK 1.1.1 has the following new capabilities and refinements:

- Updated the following third-party libraries and applications:
  - FFmpeg
  - libxml2
  - Eclipse Mosquitto
  - opencv
  - OpenSSL
  - protobuf
  - sqlite
  - usb
- Added support for the Raspberry Pi 4 Model B hardware platform (Revision 1.5).

KasperskyOS Community Edition SDK 1.1 has the following new capabilities and refinements:

- Added support for working with an I2C bus in master device mode.
- Added support for working with an SPI bus in master device mode.
- Added support for USB HID devices.
- Added support for Symmetric Multiprocessing (SMP).
- Expanded capabilities for device profiling: added iperf library and counters that track system parameters.
- Added PCRE library and usage example.
- Added SPDLOG library and usage example.
- Added MessageBus component and usage example.
- Added dynamic code analysis tools (ASAN, UBSAN).

KasperskyOS Community Edition SDK 1.0 has the following new capabilities and refinements:

- Added support for the Raspberry Pi 4 Model B hardware platform.
- Added SD card support for the Raspberry Pi 4 Model B hardware platform.
- Added Ethernet support for the Raspberry Pi 4 Model B hardware platform.
- Added GPIO port support for the Raspberry Pi 4 Model B hardware platform.
- Added network services for DHCP, DNS, and NTP and usage examples.
- Added library for working with the MQTT protocol and usage examples.

About KasperskyOS Community Edition SDK

KasperskyOS Community Edition SDK is a publicly available version of the Software Development Kit (SDK) of KasperskyOS designed to help you master the basic principles of application development for KasperskyOS. KasperskyOS Community Edition SDK will let you see how the concepts rooted in KasperskyOS actually work in practical applications. KasperskyOS Community Edition SDK includes sample applications with source code, detailed explanations, and instructions and tools for building applications.

KasperskyOS Community Edition SDK will help you:

- Learn the principles and techniques of Secure by Design development based on practical examples.
- Explore KasperskyOS as a potential platform for implementing your own projects.
- Prototype solutions (primarily intended for embedded systems and the Internet of things (IoT)) based on KasperskyOS.
- Port applications/components to KasperskyOS.
- Explore security issues in software development.

KasperskyOS Community Edition SDK lets you develop applications in the C, C++ and Rust languages.

You can get the KasperskyOS Community Edition SDK and find the appropriate documentation for it in the KasperskyOS for developers section of the website.

Distribution kit

The KasperskyOS SDK is a set of software tools for creating KasperskyOS-based solutions.

The KasperskyOS SDK is installed to a computer running the Ubuntu GNU/Linux® operating system.

> KasperskyOS Community Edition SDK is supplied as a deb package for running in the QEMU emulator as well as separate deb packages for each of the supported hardware platforms.

The distribution kit of KasperskyOS Community Edition SDK includes the following:

- Deb package for installation of KasperskyOS Community Edition SDK, including:
  - Image of the KasperskyOS kernel
  - Development tools (Clang and rustc cross compilers, LD linker, GDB debugger, QEMU emulator, and accompanying tools)
  - Tools and scripts (such as source code generators and build system scripts)
  - A set of libraries that provide partial compatibility with the POSIX standard

    - Drivers
    - System programs (for example, virtual file system)
    - Usage examples for components of KasperskyOS Community Edition SDK
    - End User License Agreement
    - Information about third-party code (Legal Notices)
- KasperskyOS Community Edition SDK Developer's Guide (Online Help)
- Release Notes

The following components included in the KasperskyOS Community Edition SDK distribution kit are the Runtime Components as defined by the terms of the License Agreement:

- Image of the KasperskyOS kernel.

All the other components of the distribution kit are not the Runtime Components. Terms and conditions of the use of each component can be additionally defined in the section "Information about third-party code".

System requirements

KasperskyOS Community Edition SDK is supplied as a deb package for running in the QEMU emulator as well as separate deb packages for each of the supported hardware platforms.

To install and use KasperskyOS Community Edition SDK, and to run examples on a virtual machine in a QEMU emulator (provided in the SDK), the following is required:

1. **Operating system:** Ubuntu GNU/Linux 24.04 "Noble Numbat". It is recommended to use a Docker container to deploy the operating system image.
2. **CPU:** processor with an x86-64 architecture.
3. **RAM**: it is recommended to have at least 4 GB of RAM for convenient use of the build tools.
4. **Disk space**: at least 5 GB of free space in the `/opt` folder (depending on the solution being developed).

To run examples on the Raspberry Pi hardware platform, the following is required:

- Raspberry Pi 4 Model B (Revision 1.4, 1.5) with 4 or 8 GB of RAM
- MicroSD card with at least 2 GB
- USB-UART converter that supports a speed of 115,200 baud

To run examples on the hardware platform OrangePi RV2, the following is required:

- OrangePi RV2 (RV64 architecture) Revision 1.1 with 8 GB of RAM
- MicroSD card with at least 2 GB
- USB-UART converter that supports a speed of 115,200 baud

Included third-party libraries and applications

To simplify the application development, KasperskyOS Community Edition SDK also includes the following third-party libraries and applications:

- **flex (v.2.6.2)** is a lexical analyzer generator.

  Documentation: https://github.com/westes/flex

- **pkg-config-lite (v.0.28)** is a tool that provides an interface for getting information about the libraries installed in the system (such as the version and parameters for the C/C ++ compiler and linker).

  Documentation: https://sourceforge.net/projects/pkgconfiglite

- **CMake (v.3.29.0)** is a cross-platform software tool that automatically builds software from source code.

  Documentation: https://cmake.org/documentation

- **autoconf-archive (v.2022.09.03)** is a set of macros for the Autoconf tool, which creates configuration scripts for automatically configuring and building software from source code.

  Documentation: https://www.gnu.org/software/autoconf-archive

- **Automake (v.1.16.4)** is a tool that generates standard `Makefile.in` files for automatically configuring and building software from source code.

  Documentation: https://www.gnu.org/software/automake

- **Autoconf (v.2.72)** is a tool that generates `configure` scripts for automatically configuring and building software from source code.

  Documentation: https://www.gnu.org/software/autoconf

- **Libtool (v.2.4.7)** is a generic library support script that conceals the complexity of using shared libraries behind a consistent, portable interface.

  Documentation: https://www.gnu.org/software/libtool

- **Binutils (v.2.46.1)** is a set of tools for working with binary files that includes an assembler, linker, archiver, and other tools.

  Documentation: https://www.gnu.org/software/binutils

- **Bison (v.3.5.4)** is a general-purpose syntax analyzer generator that converts an annotated context-free grammar into an LR or GLR parser employing LALR(1) parser tables.

  Documentation: https://www.gnu.org/software/bison

- **QEMU (v.9.2.4)** is a program that emulates hardware of various platforms.

  Documentation: https://www.qemu.org/docs/master

- **Automated Testing Framework (ATF) (v.0.20)** is a set of libraries for writing tests for programs in C, C++ and POSIX shell.

  Documentation: https://github.com/jmmv/atf

- **Boost (v.1.85.0)** is a set of class libraries that utilize C++ language functionality and provide a convenient cross-platform, high-level interface for concise coding of various everyday programming subtasks (such as working with data, algorithms, files, threads, and more).

  Documentation: https://www.boost.org/doc

- **nlohmann_json (v.3.9.1)** is the library for working with JSON format.

  Documentation: https://github.com/nlohmann/json

- **fmt (v.11.0.2)** is an open-source formatting library.

  Documentation: https://fmt.dev/latest/index.html

- **Guidelines Support Library (GSL) (v.2.1.0)** is a library containing functions and types that are suggested for use by the C++ Core Guidelines maintained by the Standard C++ Foundation.

  Documentation: https://github.com/microsoft/gsl

- **GoogleTest (v.1.14.0)** is a C++ code testing library.

  Documentation: https://google.github.io/googletest

- **iperf (v.3.19.1)** is a network performance testing library.

  Documentation: https://software.es.net/iperf

- **json-schema-validator (v.2.3.0)** is a library designed for validating data in JSON format according to defined JSON schemas.

  Documentation: https://github.com/pboettch/json-schema-validator

- **libffi (v.3.2.1)** is a library providing a C interface for calling previously compiled code.

  Documentation: https://github.com/libffi/libffi

- **jsoncpp (v.1.9.4)** is a library for working with JSON format.

  Documentation: https://github.com/open-source-parsers/jsoncpp

- **libpcap (v.1.10.5)** is a library for developing programs that can capture, filter, and analyze network traffic in UNIX-like systems.

  Documentation: https://www.tcpdump.org/index.html#documentation

- **libunwind (v.1.6.2)** is a library for handling exceptional situations and implementing a mechanism for backtracing the stack of function calls when a process crashes.

  Documentation: https://www.nongnu.org/libunwind/docs.html

- **libxml2 (v.2.15.4)** is a library for working with XML.

  Documentation: http://xmlsoft.org

- **Mbed TLS (v.3.6.6)** is a library that implements cryptographic protocols such as TLS/SSL and DTLS, and algorithms for encryption, hashing, and authentication.

  Documentation: https://mbed-tls.readthedocs.io/en/latest

- **Eclipse Mosquitto (v2.0.18)** is a message broker that implements the MQTT protocol.
  Documentation: https://mosquitto.org/documentation
- **NTP (v.4.2.8P15)** is a library for working with the NTP time protocol.
  Documentation: http://www.ntp.org/documentation.html
- **OpenSSL (v.3.5.7)** is a full-fledged open-source encryption library.
  Documentation: https://www.openssl.org/docs/
- **pcre (v.8.44)** is a library for working with regular expressions.
  Documentation: https://www.pcre.org/current/doc/html
- **sqlite (v.3.53.2)** is a library for working with databases.
  Documentation: https://www.sqlite.org/docs.html
- **Zlib (v.1.3.2)** is the data compression library.
  Documentation: https://zlib.net/manual.html
- **usb (v.14.0.0)** is a library for working with USB devices.
  Documentation: https://github.com/freebsd/freebsd-src/tree/release/13.0.0/sys/dev/usb
- **libevdev (v.1.12.1)** is a library for working with evdev peripheral devices.
  Documentation: https://www.freedesktop.org/software/libevdev/doc/latest
- **dhcpcd (v.9.4.1)** is a DHCP/DHCPv6 client intended for automatic configuration of network settings on the client side.
  Documentation: https://github.com/NetworkConfiguration/dhcpcd
- **Lwext4 (v.1.0.0)** is a library for working with the ext2/3/4 file systems.
  Documentation: https://github.com/gkostka/lwext4.git
- **clang (v.17.0.6)** is a C/C++ compiler.
  Documentation: http://llvm.org/
- **corrosion-rs/corrosion (v.0.2.2)** is a tool for integrating Rust into an existing CMake project.
  Documentation: https://github.com/corrosion-rs/corrosion
- **google/re2 (v.2022-02-01)** is a library for working with regular expressions.
  Documentation: https://github.com/google/re2
- **mtheall/ftpd (v.2.3.0)** is a File Transfer Protocol (FTP) server.
  Documentation: https://github.com/mtheall/ftpd
- **python (v.3.14.0)** is a general-purpose, high-level programming language.
  Documentation: https://github.com/python/cpython/
- **rust (v.1.77.2)** is a general-purpose, multi-paradigm compiled programming language.
  Documentation: https://www.rust-lang.org/

- **wpa_supplicant (v.2.11)** is a cross-platform, open implementation of the IEEE 802.11 standard.
  Documentation: https://w1.fi
- **yaml/libyaml (v.0.2.5)** is a library for working with YAML.
  Documentation: https://yaml.org/
- **Immediate-Mode-UI/Nuklear** (v4.12.7)
  Documentation: https://immediate-mode-ui.github.io/Nuklear/
- **Skia** (v.m112)
  Documentation: https://skia.org/docs/
- **zxing-cpp/zxing-cpp** (v.2.3.0)
  Documentation: https://github.com/zxing-cpp/zxing-cpp

See also Information about third-party code.

Drivers included in the SDK

The drivers included in KasperskyOS Community Edition SDK for the supported hardware platforms are listed in the table below.

Drivers for the Raspberry Pi 4 Model B and OrangePi RV2 are supplied in binary format.

See also Developing drivers for KasperskyOS.

Drivers provided in KasperskyOS Community Edition SDK

Driver	Raspberry Pi 4 B	OrangePi RV2
Bcm2711MboxArmToVc	In binary format	No
Bluetooth	In binary format	No
BSP	In binary format	In binary format
Camera	In binary format	No
Ethernet	In binary format	In binary format
FrameBuffer	In binary format	No
GPIO	In binary format	In binary format
I2C	In binary format	In binary format
MBIM	In binary format	No
Pinctrl	In binary format	In binary format
SDCard	In binary format	In binary format
SPI	In binary format	In binary format
UART	In binary format	In binary format
USB (HID and Mass storage)	In binary format	In binary format
USB-UART	In binary format	No
Watchdog	In binary format	No
WiFi	In binary format	No

Limitations and known issues

1. When starting two or more programs with the same EDL description, the running processes by which they are identified must have different names. To ensure the uniqueness of the process name, the ExecutionManager component adds a suffix with a specific number.

2. The system stops if no running programs remain, or if one of the driver program threads has been terminated, whether normally or abnormally.

3. When connecting different USB devices, the `device-id` counter increases by a different value.

4. Some examples contain errors in the boot and execution logs that do not affect the functionality.

5. The Rust language, the Toybox component, the Kaspersky Control Flow Monitor (KCFM) component, and the neovim debug plug-in are included in experimental mode, so their full functionality cannot be guaranteed.

6. Full porting of third-party libraries to KasperskyOS cannot be guaranteed.

7. Network packets may be delayed after KasperskyOS startup due to operation of the STP protocol.

8. The libc library API supported by VFS has a limit of 30 VFS clients and 5 threads per client.

9. For a stack backtrace, the kernel may not always be able to display the names of functions imported from dynamic libraries. It is recommended to use the `gdb` debugger or the `Dump Collector` component to analyze the stack.

10. After finishing examples that use the network, running processes of an example may remain in the system in QEMU by using the Ctrl+C key combination.

11. Some GPIO pins of hardware devices may contain external pull-up resistors. If these types of resistors are present, weaker internal resistors will not allow these pins to be pulled to 0.

12. Due to the limitations of miniUART on the Raspberry Pi 4 Model B hardware platform, the `pl011 uart` port is used as a diagnostic port by default, which disables Bluetooth. To enable Bluetooth, you must delete the following lines from the `rpi4b_prepare_boot_partition.sh` file before building the examples:

    - `dtoverlay=disable-bt`
    - `earlycon=pl011,mmio32,0xfe201000`

13. On the Raspberry Pi 4 Model B hardware platform, the SPI driver has a maximum frequency limit of 50 MHz.

14. On the Raspberry Pi 4 Model B hardware platform, there is a periodic delay in processing the camera frame queue (up to 2 seconds). Automatic restart of frozen transactions has been implemented, but single frame drops are possible when working with high-resolution cameras.

15. With intensive logging through the kernel, the solution may slow down or fail to start due to timeouts. Potential solution: increase all timeouts.

16. When you turn off a USB device by using the `UsbSetPowerMode(..., USB_POWER_MODE_OFF)` function, you cannot turn it back on by using the `UsbSetPowerMode(..., USB_POWER_MODE_ON)` function.

17. For the OrangePi RV2 hardware platform, ISA is implemented without support for "v" vector instructions. Using such instructions will result in errors.

18. On the OrangePi RV2 hardware platform, eMMC cards may work incorrectly in the microSD slot. This is due to the specific features of the Ky X1 chip, which supports the operations of eMMC modules only on the eMMC connector of the board.

19. The OrangePi RV2 hardware platform does not support a GDB server. You are advised to use JTAG for debugging.

20. The reboot command does not work on the OrangePi RV2 hardware platform in the uart_toybox example. You can use the hardware RESET button to reboot the device.

21. Upgrading from KasperskyOS Community Edition SDK 1.4 to 2.0 without changing the Ubuntu GNU/Linux 22.04 "Jammy Jellyfish" operating system to the Ubuntu GNU/Linux 24.04 "Noble Numbat" operating system is not supported.

Critical changes in version 2.0

Due to modifications made to SDK components in version 2.0, you may need to make changes to the application code that was developed using KasperskyOS Community Edition SDK version 1.4 before using that code with the new version.

The following critical changes were made to SDK components in version 2.0.

General changes

1. The NameServer component has been removed from the SDK. To dynamically create IPC channels, use the API of the DCM system program.

2. The klog component has been removed from the SDK. Messages from Kaspersky Security Module are now displayed in the kernel console. You are advised to use logrr to collect them.

3. The PackageManager component no longer uses a database to store information about installed KPA packages; the arguments defining the path to the database in `create_package_manager_entity` and in the package preinstallation functions (for example, kl_install_packages) have been removed.

4. All tools in `sys/rump/libcliapp` have been renamed from `${name}_entity` to `${name}_service`, and their CMake components and targets have been updated (for example, `PING_ENTITY` -> `PING_SERVICE`, `rump::ping_entity` -> `rump::ping_service`). When using `find_package(rump REQUIRED COMPONENTS ...)`, you need to specify these new names.

5. The container mechanism associated with FsImageManager has been removed from the ExecutionManager component. The privateStorage key remains in the `KPA package manifest` for backward compatibility, but it will be ignored by the PackageManager and ExecutionManager system programs.

6. The `EntityLauncher.StartEntity` interface has been removed from the API of the BlobContainer system program.

7. The `--audit` argument has been removed from nk-psl-gen-c.

Changes in the API of the VFS system program

- The `_vfs_open` method, `_vfs_socket` method, and similar methods in `vfs::client` have been changed: they will now return `Handle` from the internal table, not from the kernel table. To get a kernel handle, use `_vfs_get_kernel_handle`.
- VFS combines the `vfs_read`, `vfs_pread`, and `vfs_readv` methods into one universal `vfs_read`.

Changes in the libkos library API

- The KnProfilerGetCounters*() API has been removed. It is recommended to switch to the KnStat*() API.
- The obsolete read_mdl / write_mdl interfaces have been removed. It is recommended to use the MDL mechanism.
- RTC PL031 support has been removed: after starting the system, KnGetSystemTime will return 1970-01-01T00:00:00 (or the value specified via `-rtc` in QEMU), KnGetRtcInfo returns rcUnimplemented; the time can still be set via KnSetSystemTime(), and the kernel will return it correctly until restarted.
- The names of the public methods of the container API in the libkos library now follow a consistent naming style.
- If a deferred memory allocation error occurs, the kernel will now wait for free pages to appear instead of throwing an `EXCEPTION_OVERCOMMIT` error.

Changes in the API for driver development

- The bsp driver for Raspberry Pi 4B now receives device configurations from the Device Tree, not from a static table, and the names of devices and configurations now correspond to DT.
- The convention of naming objects in pinctrl has been simplified for Raspberry Pi 4B: optional prefixes have been removed (for example, `kl.drivers.pinctrlModule/pmic@20` -> `pmic@20`, `kl.drivers.pinctrlGpioRange/gpio0` -> `gpio0`).
- The signatures of the initialization and termination functions of server components with a security interface have been changed: `*_securityIidOffset` (generated from *.edl.h) is now used instead of `*_securityIid`. Initializers have been made resistant to multiple calls. The affected libraries are kdf, blkdev (initialization of the blkdev server is encapsulated in BlkDevServerInit), pcie, and usb.

Training scenarios

This section presents step-by-step training scenarios for working with KasperskyOS Community Edition SDK. We recommend that you start with the Obtaining and installing the SDK scenario.

Obtaining and installing the SDK

After sequentially following the instructions in this section, you can start working with KasperskyOS Community Edition SDK and the KasperskyOS SDK Extension for Visual Studio Code. You will need a computer that has the Ubuntu GNU/Linux 24.04 "Noble Numbat" operating system installed (see more details about the system requirements).

Getting KasperskyOS Community Edition SDK

To download a deb package containing KasperskyOS Community Edition SDK:

1. Go to the website https://os.kaspersky.com/development/#start.
2. Click the **Download** button.
3. Carefully read the terms and conditions of the KasperskyOS Community Edition SDK End User License Agreement, fill out the form, and click **Send**.

   A link to download the SDK will be sent to your email address.
4. Follow the link to download the SDK.

   KasperskyOS Community Edition SDK is supplied as a deb package for running in the QEMU emulator as well as separate deb packages for each of the supported hardware platforms.
5. Click the link to download the current version of the deb package to be run in the QEMU emulator and select the directory where the file will be saved. For example: `~/Downloads`.
6. Wait for the download to complete.

Installing the SDK

> You are advised to install the SDK and work with it in a Docker container.

1. Make sure that the Docker software is installed and running.
2. To download the official Docker image of the Ubuntu GNU/Linux 24.04 (Noble Numbat) operating system from the public Docker Hub repository, run the following command:

   ```
   docker pull ubuntu:24.04
   ```

3. To run the image, run the following command:

   ```
   docker run --net=host  --user root --privileged -it --rm ubuntu:24.04 bash
   ```

4. Open a second terminal and get the ID of the container with the running image by using the `docker ps` command.
5. In the same terminal, copy the deb package for installing the KasperskyOS Community Edition SDK into the container with the obtained ID by running the following command:

   ```
   docker cp KasperskyOS-Community-Edition-<version>.deb <container_id>:/opt
   ```

6. In the first terminal where the Docker container was started, install the KasperskyOS Community Edition SDK:

   ```
   sudo apt update && sudo apt install ~/Downloads/KasperskyOS-Community-Edition-Qemu-<version>_<platform>.deb
   ```

   The SDK will be installed in the `/opt/KasperskyOS-Community-Edition-Qemu-<version>` directory.

7. To conveniently work with tools provided in the SDK, the path to the executable files of these tools in `/opt/KasperskyOS-Community-Edition-<platform>-<version>/toolchain/bin` must be added to the `PATH` environment variable in the same terminal. To do so, the following commands must be executed:

   ```
   cd /opt/KasperskyOS-Community-Edition-Qemu-<version>/common/
   source set_env.sh
   ```

   In addition to changing the `PATH` environment variable, the script also defines the `KOSCEVER` and `KOSCEDIR` environment variables that contain the version and absolute path to the KasperskyOS Community Edition SDK, respectively. Use of these environment variables allows to determine the SDK installation path, and to verify that the solution version matches the SDK version.

   If the environment variables were successfully defined, the `printenv | grep KOSCE` command will print the paths contained in the variables.

> Environment variables are recognized only in the current terminal. When using a different terminal, the variables must be defined again.

Related sections:
Using a Docker container
Installation and removal

Quick Start. Introduction to the SDK

Prior to studying this scenario, you need to obtain and install the SDK.

You are advised to consecutively carry out the instructions provided in this section to quickly familiarize yourself with the basics of using the KasperskyOS Community Edition SDK.

Checking out the "hello" example and running the example in QEMU

1. Copy the directory containing the source code of the "hello" example to the home directory by running the following command:

   ```
   cp -r /opt/KasperskyOS-Community-Edition-Qemu-<version>/examples/hello/ ~/kasperskyos_hello
   ```

   > This step is required because the examples must be built in a directory to which you have write-access rights.

2. Go to the directory containing the example code by running the following command:

   ```
   cd ~/kasperskyos_hello
   ```

3. Review the contents of the directory. The source code of the "hello" example consists of the following files:

```
.
├── CMakeLists.txt               # A CMakeLists.txt root file with the general build instructions for the
entire
│                                # KasperskyOS-based solution.
├── einit
│   ├── CMakeLists.txt           # CMakeLists.txt file for building a KasperskyOS boot image.
│   └── src
│       ├── initial_install.json # The file contains a list of KPA packages that must be installed when
KasperskyOS is started for the first time. Used by the ServiceManager system program.
│       ├── security.psl         # Security policy description, which is used by the build system to
automatically generate a security module.
│       └── system_services.json # This file contains a list of installed programs that must be started each
time KasperskyOS is started. Used by the ServiceManager system program.
├── hello
│   ├── CMakeLists.txt       # CMakeLists.txt files for building the hello application and packing it into a
KPA package.
│   └── src
│       └── hello.c              # Source code of the hello application.
├── README.md                    # Instructions for building the example.
└── resources
    └── xdl
        └── hello
            └── Hello.edl        # Description of the solution component in EDL. Required to create a solution
security policy.
                                 # The Hello program has the minimum possible description: an EDL file in which
                                 # the process class name is specified.
```

Examples from KasperskyOS Community Edition SDK are built using the CMake build system into a KasperskyOS-based solution image, which is subsequently run in the emulator or on the hardware platform.

The kernel image and the set of KasperskyOS system programs are delivered as part of the KasperskyOS Community Edition SDK. Application software included in the solution is built from the example source code files using the `CMakeLists.txt` build files.

The "hello" example consists of a single "hello" application that prints the text `"Hello world!"` to the standard error stream.

The rest of the files in the example directory are needed to build the "hello" program, build the boot image, and create a solution security policy.

4. To build the example, run the following commands:

```
cmake -B build -D CMAKE_TOOLCHAIN_FILE="$KOSCEDIR/toolchain/share/toolchain-aarch64-kos.cmake"
cmake --build build --target sim
```

After the solution image is built, the example starts in the QEMU emulator in the current terminal window.

You will see the KasperskyOS boot log and the message `"Hello world!"` that is printed to the standard error stream by the hello program.

Related sections:
KasperskyOS Developer's Quick Start Guide

hello example

Building the examples

Running examples on QEMU

Working with KPA packages

Working with the extension for Visual Studio Code

Prior to studying this scenario, you need to obtain and install the SDK.

The KasperskyOS SDK Extension for Visual Studio Code integrates the KasperskyOS Community Edition SDK with the source code editor known as Visual Studio Code.

> The KasperskyOS SDK Extension for Microsoft Visual Studio Code is contained only in the deb package to be run in the QEMU emulator.

By sequentially following the instructions in this section, you will run simple examples in the C and Rust languages in the QEMU emulator while using the familiar Microsoft Visual Studio Code development environment.

Checking out the hello example in C

1. Copy the directory containing the source code of the "hello" example to the home directory by running the following command:

   ```
   cp -r /opt/KasperskyOS-Community-Edition-Qemu-<version>/examples/hello/ ~/kasperskyos_hello
   ```

   > This step is required because the examples must be built in a directory to which you have write-access rights.

2. Open the directory containing the example code in the Visual Studio Code editor.

3. Review the contents of the directory. The source code of the "hello" example consists of the following files:

```
.
├── CMakeLists.txt              # A CMakeLists.txt root file with the general build instructions for the
entire
│                               # KasperskyOS-based solution.
├── einit
│   ├── CMakeLists.txt          # CMakeLists.txt file for building a KasperskyOS boot image.
│   └── src
│       ├── initial_install.json # The file contains a list of KPA packages that must be installed when
KasperskyOS is started for the first time. Used by the ServiceManager system program.
│       ├── security.psl        # Security policy description, which is used by the build system to
automatically generate a security module.
│       └── system_services.json # This file contains a list of installed programs that must be started each
time KasperskyOS is started. Used by the ServiceManager system program.
├── hello
│   ├── CMakeLists.txt      # CMakeLists.txt files for building the hello application and packing it into a
KPA package.
│   └── src
│       └── hello.c             # Source code of the hello application.
├── README.md                   # Instructions for building the example.
└── resources
    └── xdl
        └── hello
            └── Hello.edl       # Description of the solution component in EDL. Required to create a solution
security policy.
```

```
                                # The Hello program has the minimum possible description: an EDL file in which
                                # the process class name is specified.
```

Examples from KasperskyOS Community Edition SDK are built using the CMake build system into a KasperskyOS-based solution image, which is subsequently run in the emulator or on the hardware platform.

The kernel image and the set of KasperskyOS system programs are delivered as part of the KasperskyOS Community Edition SDK. Application software included in the solution is built from the example source code files using the `CMakeLists.txt` build files.

The "hello" example consists of a single "hello" application that prints the text `"Hello world!"` to the standard error stream.

The rest of the files in the example directory are needed to build the "hello" program, build the boot image, and create a solution security policy. The scenario for building and running the example is described below in the "Working with the example in the extension for Visual Studio Code" subsection.

Checking out examples in the Rust language

1. Copy the directory containing the source code of the "hello_from_rust " example to the home directory by running the following command:

   ```
   cp -r /opt/KasperskyOS-Community-Edition-Qemu-<version>/examples/hello_from_rust/ ~/kasperskyos_hello_from_rust
   ```

   > This step is required because the examples must be built in a directory to which you have write-access rights.

2. Open the directory containing the example code in the Visual Studio Code editor.

3. Review the contents of the directory. The source code of the "hello_from_rust " example consists of the following files:

```
.
├── CMakeLists.txt              # A CMakeLists.txt root file with the general build instructions for the
entire KasperskyOS-based solution.
├── einit
│   ├── CMakeLists.txt          # CMakeLists.txt file for building a KasperskyOS boot image.
│   └── src
│       ├── initial_install.json # The file contains a list of KPA packages that must be installed when
KasperskyOS is started for the first time. Used by the ServiceManager system program.
│       ├── security.psl        # Security policy description, which is used by the build system to
automatically generate a security module.
│       └── system_services.json # This file contains a list of installed programs that must be started each
time KasperskyOS is started. Used by the ServiceManager system program.
├── hello
│   ├── build.rs             # Custom build script in the Rust language.
│   ├── Cargo.toml           # Main configuration file and project manifest in the Rust language.
│   ├── CMakeLists.txt       # CMakeLists.txt files for building the hello application.
│   └── src
│       └── main.rs          # Source code of the hello application
├── README.md
└── resources
    └── xdl
        └── hello_from_rust
            └── Hello.edl    # Description of the solution component in EDL. Required to create a solution
security policy.
                             # The Hello program has the minimum possible description: an EDL file in which
                             # the process class name is specified.
```

The security policy file named security.psl is similar to the corresponding files from C-language projects. There are no differences for programming languages because the security policy does not depend on the language in which the project is implemented.

Building and running the example is similar to the C hello example. The scenario for building and running the example is described below in the "Working with the example in the extension for Visual Studio Code" subsection.

4. Check out the following example in Rust. Copy the directory containing the source code of the "hello_corrosion" example to the home directory by running the following command:

```
cp -r /opt/KasperskyOS-Community-Edition-Qemu-<version>/examples/hello_corrosion/ ~/kasperskyos_hello_corrosion
```

5. Open the directory containing the example code in the Visual Studio Code editor.

6. This example is similar to the hello_from_rust example, except that the corrosion_import_crate function is added to the CMakeLists.txt file, which integrates the program developed in Rust into the CMake project. This function compiles the hello executable file that will be included in the KasperskyOS-based solution. For more details, refer to hello_corrosion example.

Working with the example in the extension for Visual Studio Code

The KasperskyOS SDK Extension for Visual Studio Code activates when it automatically detects a KasperskyOS-based solution project in the open workspace.

> To manually activate the extension in the directory containing the example, run the command `KOS: Activate extension in this directory` in the Visual Studio Code command palette.

The activated extension adds buttons to the lower panel of the Visual Studio Code editor:



Extension buttons in the lower panel of Visual Studio Code

1. Make sure that the directory containing the "hello" example code is open in the Visual Studio Code editor and the extension is activated.

2. Click the **Run CMake** button ( ) in the lower panel of Visual Studio Code.

   The extension runs scripts to configure the current project's build directory for the selected build architecture.

3. Click the select build target button (**all** by default) in the lower panel of Visual Studio Code and select the **sim** build target.

4. Click the **Build the selected target** button ( ) in the lower panel of Visual Studio Code.

   After the solution image is built, the example starts in the QEMU emulator in the current terminal window of Visual Studio Code.

   You will see the KasperskyOS boot log and the message `"Hello world!"` that is printed to the standard error stream by the hello program.

Checking out the security policy example

A security policy for a KasperskyOS-based solution lets you allow or deny interactions between processes and the KasperskyOS kernel.

To create a basic description of the security policy in the hello example:

1. Press **F1**.

2. Run the command `KOS: Generate policy file for current project` in the command line that appears in the upper part of the window.

3. Select `Grant necessary permissions` in the same window.

4. Review the contents of the basic security policy in the generated file.

You can now edit the policy to get more complex interaction rules in your solution.

Related sections:
KasperskyOS SDK Extension for Visual Studio Code
Hello example
Hello_corrosion example
Developing security policies

Running an example on the hardware platform

Prior to studying this scenario, you need to obtain and install the SDK.

By sequentially following the instructions in this section, you will run a simple example on the Raspberry Pi 4 Model B hardware platform.

To run your first program on the Raspberry Pi hardware platform, you need the following:

- Raspberry Pi 4 Model B (Revision 1.4, 1.5) with 4 or 8 GB of RAM
- MicroSD card with at least 2 GB
- USB-UART converter that supports a speed of 115,200 baud

1 Checking out the hello example

1. Copy the directory containing the source code of the "hello" example to the home directory by running the following command:

   ```
   cp -r /opt/KasperskyOS-Community-Edition-RaspberryPi4b-<version>/examples/hello/ ~/kasperskyos_hello
   ```

   > This step is required because the examples must be built in a directory to which you have write-access rights.

2. Go to the directory containing the example code by running the following command:

   ```
   cd ~/kasperskyos_hello
   ```

3. Review the contents of the directory. The source code of the "hello" example consists of the following files:

   ```
   .
   ├── CMakeLists.txt               # A CMakeLists.txt root file with the general build instructions for the
   entire
   │                                # KasperskyOS-based solution.
   ├── einit
   │   ├── CMakeLists.txt           # CMakeLists.txt file for building a KasperskyOS boot image.
   │   └── src
   │       ├── initial_install.json # The file contains a list of KPA packages that must be installed when
   KasperskyOS is started for the first time. Used by the ServiceManager system program.
   │       ├── security.psl         # Security policy description, which is used by the build system to
   automatically generate a security module.
   │       └── system_services.json # This file contains a list of installed programs that must be started each
   time KasperskyOS is started. Used by the ServiceManager system program.
   ├── hello
   │   ├── CMakeLists.txt       # CMakeLists.txt files for building the hello application and packing it into a
   KPA package.
   │   └── src
   │       └── hello.c              # Source code of the hello application.
   ├── README.md                    # Instructions for building the example.
   └── resources
       └── xdl
           └── hello
               └── Hello.edl        # Description of the solution component in EDL. Required to create a solution
   security policy.
                                    # The Hello program has the minimum possible description: an EDL file in which
                                    # the process class name is specified.
   ```

   Examples from KasperskyOS Community Edition SDK are built using the CMake build system into a KasperskyOS-based solution image, which is subsequently run in the emulator or on the hardware platform.

The kernel image and the set of KasperskyOS system programs are delivered as part of the KasperskyOS Community Edition SDK. Application software included in the solution is built from the example source code files using the `CMakeLists.txt` build files.

The "hello" example consists of a single "hello" application that prints the text `"Hello world!"` to the standard error stream.

The rest of the files in the example directory are needed to build the "hello" program, build the boot image, and create a solution security policy.

2 Building the hello example to run on Raspberry Pi

1. To build the example, run the following commands:

```
cmake -B build -D CMAKE_TOOLCHAIN_FILE="$KOSCEDIR/toolchain/share/toolchain-aarch64-kos.cmake"
cmake --build build --target sd-image
```

The build process creates a file system image for a bootable SD card. The following is loaded into the file system image:

- KasperskyOS-based solution image
- U-Boot bootloader that starts the example
- firmware for Raspberry Pi 4 B.

The source code for the U-Boot bootloader and firmware can be downloaded from the internet.

The file system image file `hdd.img` is saved in the `~/kasperskyos-hello/build` directory.

2. To prepare a bootable SD card for the Raspberry Pi 4 B, connect the SD card to your computer and run the following command (where [X] is the last character in the name of the block device for the SD card):

```
$ sudo dd of=/dev/sd[X] if=~/kasperskyos-hello/build/hdd.img status=progress bs=64K conv=fsync
```

3 Preparing Raspberry Pi 4 B and running examples

To prepare your Raspberry Pi 4 B to run the example and see the output of the example on your computer, do the following:

1. Connect the pins of the FT232 USB-UART converters to the corresponding GPIO pins of Raspberry Pi 4 B (see the figure below). Because debugging is not necessary, all you need to do is connect one USB-UART converter for output.

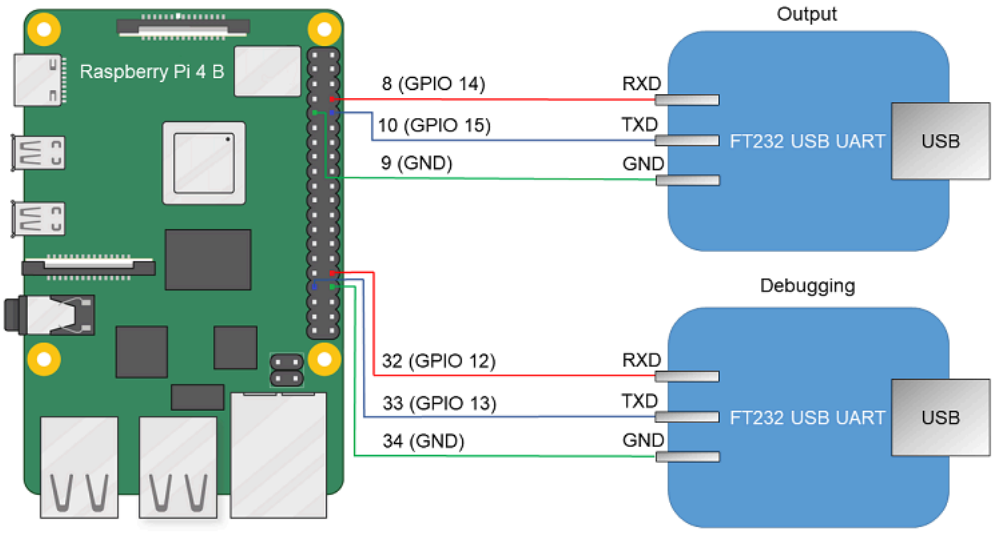


Diagram for connecting USB-UART converters and Raspberry Pi 4 B

2. Connect the computer's USB ports and the USB-UART converters.

3. Install PuTTY or another equivalent program. Configure the settings as follows: `bps = 115200`, `data bits = 8`, `stop bits = 1`, `parity = none`, `flow control = none`. Define the USB port connected to the USB-UART converter used for receiving output from Raspberry Pi 4 B.

To run an example on a Raspberry Pi 4 B:

1. Connect the bootable SD card to the Raspberry Pi 4.

2. Supply power to the Raspberry Pi 4 B and wait for the example to run.

   After starting the example, you will see the KasperskyOS boot log and the message `"Hello world!"` that is printed to the standard error stream by the hello program.

Related sections:
Preparing Raspberry Pi 4 B to run examples
Running examples on Raspberry Pi 4 B or OrangePi RV2

Debugging an example in Neovim

Before studying the scenario, get and install the SDK and define the environment variables.

By sequentially following the instructions in this section, you will debug the example using a breakpoint.

Preparing to work with Neovim and the vimspector plug-in

1. Install Neovim:

```
sudo apt install neovim
```

2. Create a directory for Neovim configurations:

```
mkdir -p ~/.config/nvim
```

3. Install the Vim-plug extension manager for Neovim:

```
curl -fLo ~/.local/share/nvim/site/autoload/plug.vim --create-dirs \
  https://raw.githubusercontent.com/junegunn/vim-plug/master/plug.vim
```

4. Install the python3-pynvim package, which serves as the client and plug-in host for Neovim:

```
apt install python3-pynvim
```

5. Create a Neovim configuration file:

```
nvim ~/.config/nvim/init.vim
```

6. Add the following configuration to the file:

```
call plug#begin('~/.vim/plugged')
Plug 'puremourning/vimspector'
call plug#end()
let g:vimspector_enable_mappings = 'HUMAN'
```

7. Install the plug-in using the `:PlugInstall` command in the Neovim terminal.

   If the "not an editor command pluginstall" error occurs, run the `:source ~/.config/nvim/init.vim` command.

   As a result, the following type of message should appear in the Neovim terminal:

```
 Updated. Elapsed time: 2.336718 sec.
[=]

- Finishing ... Done!
- vimspector: Already up to date.
```

   To exit the window, enter `:q` and press **`Enter`**.

Preparing to start debugging

1. Copy the directory containing the ping example source code into your home directory:

```
cp -r /opt/KasperskyOS-Community-Edition-Qemu-<version>/examples/ping/ ~/kasperskyos_ping
```

2. In the client/src/client.c file, replace the loop with an infinite loop:

```
 - for (i = 0; i < 5; ++i)
 + while (1)
    {
        value = ping(value);
        value = pong(value);
    }
```

3. In the root directory of the example, create a .vimspector.json configuration file for debugging in GDB with the following contents:

```
{
  "adapters": {
    "kos-gdb": {
      "name": "kos-gdb",
      "command": [
        "/opt/KasperskyOS-Community-Edition-Qemu-<version>/toolchain/bin/debug_adapter",
        "--gdb-path",
        "/opt/KasperskyOS-Community-Edition-Qemu-<version>/toolchain/bin/aarch64-kos-gdb",
        "--transport",
        "stdio"
      ]
    }
  },

  "configurations": {
    "KOS attach (Hello)": {
      "default": true,
      "adapter": "kos-gdb",
      "filetypes": [ "c", "cpp" ],
      "configuration": {
        "request": "attach",
        "name": "Attach and debug",
        "gdbServer": "tcp::12345",
        "useExtendedRemote": true,
        "setupCommands": [
          {
            "description": "Load gdb_init",
            "text": "source  <path to project>/gdb_init",
            "ignoreFailures": false
          }
        ],
        "setupServerCommands": [
            {
            "description": "Load setup script",
            "text": "source <path to project>/setup",
            "ignoreFailures": false
          }
        ]
      }
    }
  }
}
```

Replace the `<version>` and `<path to project>` parameters with the SDK version and the path to the root directory of the project, respectively.

4. In the root directory of the example, create a setup file serving as the script for debugging in GDB with the following contents:

```
set sysroot <path to project>/kasperskyos_ping/build/einit/sysroot-symbols
set solib-search-path /opt/KasperskyOS-Community-Edition-Qemu-<version>/sysroot-aarch64-kos
set osabi auto
set scheduler-locking off
set schedule-multiple on
set follow-fork-mode parent
set follow-exec-mode same
set detach-on-fork off

monitor exec kl.ExecMgrEntity
catch exec
continue
detach inferiors 1
remove-inferiors 1

monitor exec Client.1
catch exec
continue
b main
continue
```

   Replace the `<version>` parameter with the SDK version.

5. In the root directory of the example, create a gdb_init file serving as the script for initializing debugging in GDB with the following contents:

```
shell rm -rf <path to project>/build/einit/sysroot-symbols/
shell mkdir <path to project>/build/einit/sysroot-symbols/
shell find -L <path to project>/build -type f -executable | xargs -I {} cp {} <path to
project>/build/einit/sysroot-symbols/
shell find -L <path to project>/build -type f -executable | xargs -I {} /opt/KasperskyOS-Community-Edition-
Qemu-<version>/toolchain/bin/cmake -DBINARY={} -DBUILD_ID_DIR=<path to project>/build/sysroot-symbols/ -P
/opt/KasperskyOS-Community-Edition-Qemu-<version>/toolchain/share/cmake/Modules/platform/building/build-
id.cmake 2> /dev/null
```

   Replace the `<version>` and `<path to project>` parameters with the SDK version and the path to the root directory of the example, respectively.

6. In the root directory of the example, build its image for further debugging:

```
cmake -B build -D CMAKE_TOOLCHAIN_FILE="$KOSCEDIR/toolchain/share/toolchain-aarch64-kos.cmake" -D
CMAKE_BUILD_TYPE=Debug && cmake --build build --target gdbsim
```

   As a result of the build, the following message will be displayed in the terminal:

```
[KDBG] Waiting for GDB connection infinitely
```

Working with debugging

1. In a separate terminal, open the client.c file in Neovim:

```
nvim client/src/client.c
```

2. Start vimspector by pressing debugging key (usually **F5**) or by running the `:call vimspector#Launch()` command.

   As a result, the text "`Initializing debug session KOS attach (Hello) (0)..`" should appear in the terminal window.

3. Respond to the questions that appear in the lower pane of the terminal:

```
throw: Break on Catch throw (Y/N/default: Y)? Y
catch: Break on Catch catch (Y/N/default: N)? N
rethrow: Break on Catch rethrow (Y/N/default: N)? N
```

   As a result, the execution of the program will stop at the line `fprintf(stderr, "Hello I'm client\n");`.

4. Select the line `value = ping(value);` and set a breakpoint on it by pressing **F9** or by running the `:call vimspector#ToggleBreakpoint()` command.

   As a result, this breakpoint will appear in the list of all program breakpoints. This list can be obtained by using the `:VimspectorBreakpoints` command.

```
client.c:116 Line breakpoint - VERIFIED: {}
value = ping(value);
```

   To exit the window, enter `:q` and press **Enter**.

5. Continue the execution of the program several times by pressing **F5** or by running the `:call vimspector#Continue()` command.

   As a result, the following messages will appear in the terminal with the running image:

```
Hello I'm client
[Client] Ping(777), result = 780
[Client] Pong(780), result = 783
[Client] Ping(783), result = 786
[Client] Pong(786), result = 789
...
```

6. Change the value of the "value" variable. To do so, in the section containing `vimspector.Variables[0]`, select the "value" variable, then press the **Enter + ** key combination or click **Set** in the upper-left corner of the window.

   As a result, an input line containing the text "`New Value: 809`" will appear at the bottom of the window.

7. As an example, enter 2000 and press **Enter**.

   As a result, the terminal will display the following message:

```
[Client] Ping(2000), result = 2003
[Client] Pong(2003), result = 2006
```

8. To end debugging, enter the `:qa!` command and press **Enter** in the Neovim terminal to close it.

KasperskyOS operating system overview

KasperskyOS is a specialized operating system based on a separation microkernel and security monitor.

Overview

Supported processor architectures

KasperskyOS is designed for use on hardware platforms with AArch64 and RISC-V processor architectures.

Microkernel

KasperskyOS is a microkernel operating system. The kernel provides minimal functionality, including scheduling of program execution, management of memory and input/output. The code of device drivers, file systems, network protocols and other system software is executed in user mode (outside of the kernel context).

Processes and endpoints

Software managed by KasperskyOS is executed as processes. A *process* is a running program that has the following distinguishing characteristics:

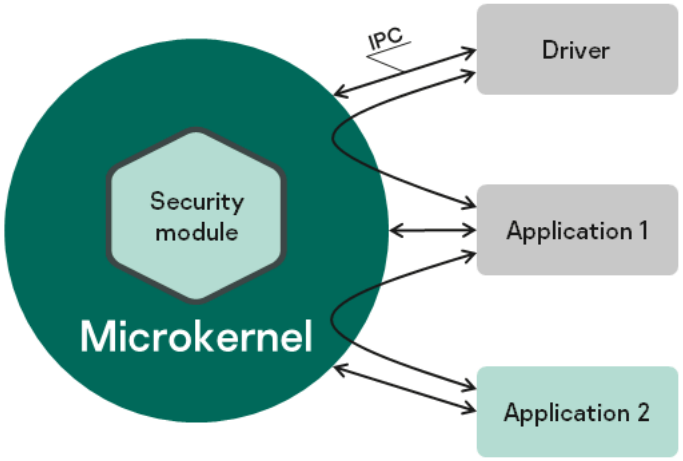
- It can provide endpoints to other processes and/or use the endpoints of other processes via the IPC mechanism.
- It uses core endpoints via the IPC mechanism.
- It is associated with security rules that regulate the interactions of the process with other processes and with the kernel.

An *endpoint* is a set of logically related methods available via the IPC mechanism (for example, an endpoint for receiving and transmitting data over the network, or an endpoint for handling interrupts).

Implementation of the MILS and FLASK architectural approaches

When developing a KasperskyOS-based system, software is designed as a set of components (programs) whose interactions are regulated by security mechanisms. In terms of security, the degree of trust in each component may be high or low. In other words, the system software includes trusted and untrusted components. Interactions between different components (and between components and the kernel) are controlled by the kernel (see the figure below), which has a high level of trust. This type of system design is based on the architectural approach known as MILS (Multiple Independent Levels of Security), which is employed when developing critical information systems.

A decision on whether to allow or deny a specific interaction is made by the Kaspersky Security Module. (This decision is referred to as the *security module decision*.) The security module is a kernel module whose trust level is high like the trust level of the kernel. The kernel executes the security module decision. This type of division of interaction management functions is based on the architectural approach known as FLASK (Flux Advanced Security Kernel), which is used in operating systems for flexible application of security policies.



Interaction between different processes and between processes and the kernel in KasperskyOS

KasperskyOS-based solution

A *KasperskyOS-based solution* (hereinafter also referred to as the *solution*) consists of system software (including the KasperskyOS kernel and Kaspersky Security Module) and applications integrated to work as part of the software/hardware system. The programs included in a KasperskyOS-based solution are considered to be *components of the KasperskyOS-based solution* (hereinafter referred to as *solution components*). Each instance of a solution component is executed in the context of a separate process.

Security policy for a KasperskyOS-based solution

Interactions between the various processes and between processes and the KasperskyOS kernel are allowed or denied according to the *KasperskyOS-based solution security policy* (hereinafter referred to as the *solution security policy* or simply the *policy*). The solution security policy is stored in the Kaspersky Security Module and is used by this module whenever it makes decisions on whether to allow or deny interactions.

The solution security policy can also define the logic for handling queries sent by a process to the security module via the *security interface*. A process can use the security interface to send some data to the security module (for example, to influence future decisions made by the security module) or to receive a security module decision that is needed by the process to determine its own further actions.

Kaspersky Security System technology

Kaspersky Security System technology lets you implement diverse security policies for solutions. You can also combine multiple security mechanisms and flexibly regulate the interactions between different processes and between processes and the KasperskyOS kernel. A Kaspersky Security Module to be used in a specific solution is created based on the solution security policy description.

Source code generators

Some of the source code of a KasperskyOS-based solution is created by source code generators from declarative descriptions. The source code of the Kaspersky Security Module and methods and types for implementing IPC (*transport code*) are generated.

The transport code is generated from the formal specifications of the solution components.

The source code of the Kaspersky Security Module is generated from the solution security policy description and formal specifications of solution components.

Architecture and startup of a KasperskyOS-based solution

KasperskyOS consists of a kernel and a set of system programs and libraries.

KasperskyOS kernel

The main component of KasperskyOS is its own proprietary kernel, which is classified as a microkernel. This means that its operating system functionality such as management of devices, file systems, and network interaction has been moved out of the kernel into system programs that run in user mode. This approach is aimed at increasing reliability by reducing the size of the kernel and distributing the operating system functions between the kernel and a set of system programs. The smaller the kernel, the fewer errors in the code that runs in privileged mode, which reduces the likelihood of an operating system malfunction. At the same time, an error in a system program can cause that program to crash, but not the entire operating system.

The kernel manages the execution of programs, memory, I/O, access to resources, and power supply of the platform, and facilitates the interaction of programs. Moreover, the kernel includes several drivers (for example, drivers for interrupt controllers and timers, UART and XHCI DbC drivers for debugging and diagnostic output) and debugging mechanisms (GDB server, performance counters).

KasperskyOS system programs

The KasperskyOS SDK contains a boot image (InitROMFS) that includes the minimum set of system programs required to run the solution and applications that have been packaged into the KPA packages.

When building a KasperskyOS-based solution using the CMake build system from the SDK, this image is always added to the solution. The system programs included in the InitROMFS image are listed in the table below.

System programs in the InitROMFS image

Program	Purpose
Einit	Starts the rest of the system programs specified in this table, and the vendor_einit system program in the VendorROMFS image.
BlobContainer	Loads executable files and dynamic libraries into RAM.
DumpCollector	Collects information about crashed processes and forwards this information to the logging system or writes it to a file.
ExecMgrEntity	Runs programs from KPA packages.
MountSettings	Sets the file system mount parameters according to the configuration specified in the `mount_settings.json` file in the VendorROMFS image.
PkgMgrEntity	Installs and removes KPA packages.
SystemSettings	Stores system parameters and information about installed KPA packages.
dcm (Dynamic Connection Manager)	Allows creating IPC channels dynamically.
vfs_storage	Provides access to file systems.

The KasperskyOS SDK also includes CMake scripts for building an additional image (VendorROMFS), which contains additional system programs and drivers required to work with the emulator or a specific hardware platform on which the solution will run.

When building a KasperskyOS-based solution using the CMake build system from the SDK, this image is created based on the descriptions of the build scenarios in the CMakeLists.txt files and then added to the solution.

The system programs included in the VendorROMFS image are listed in the table below.

System programs in the VendorROMFS image

Program(s)	Purpose
vendor_einit	Starts the rest of the system programs listed in this table.
ksm.module	The Kaspersky Security Module that is generated based on the description of a solution security policy and is integrated into the kernel at KasperskyOS boot time.
System programs each of which specific to the hardware platform and/or solution and must be started at the early stages of the KasperskyOS boot process.	Manage devices (for example, a block device driver that stores the root file system RootFS) provide functionality that is specific to a solution or hardware platform.

Kaspersky Security Module

The VendorROMFS image also includes the Kaspersky Security Module, which is generated during the image build process and integrated into the kernel at KasperskyOS boot time.

The security module is generated based on the following components of the solution security policy description:

- Description of the security policy for system programs included in the InitROMFS boot image. This description is provided as part of the SDK.
- Security policy description for system programs included in the additional VendorROMFS image. This description is generated while building this image.
- Security policy description for applications. This description is created by the SDK user. For more details, refer to Developing security policies.

rootfs file system

The rootfs root file system is mounted when KasperskyOS is started.

When building a KasperskyOS-based solution using the CMake build system from the SDK, add the following files to the rootfs root file system:

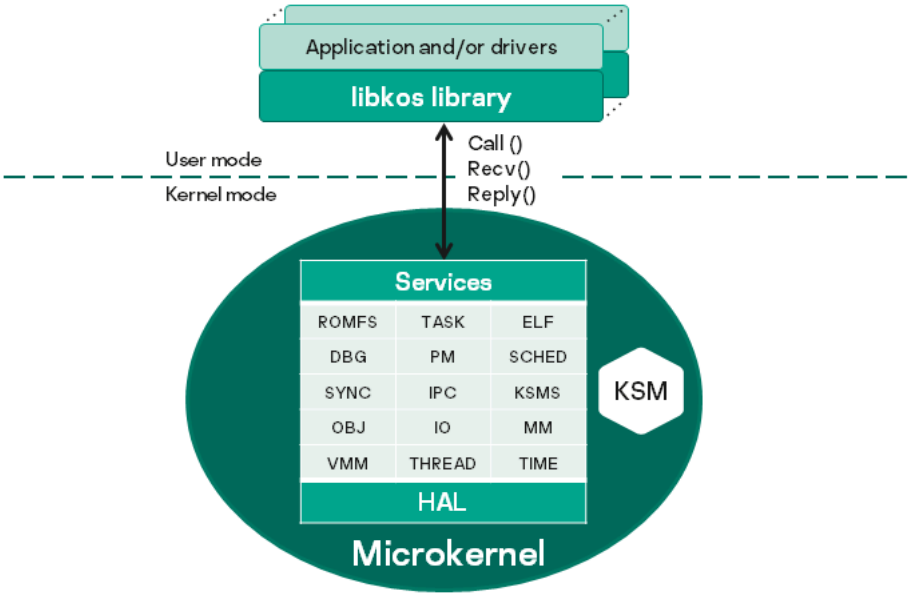
- Contents of the archive containing the libraries, files, and KPA packages necessary for the operation of system programs from the InitROMFS boot image. The archive is provided as part of the SDK.
- Files and libraries required for the operation of system programs from the VendorROMFS image.
- KPA packages of all applications included in the solution.
- Configuration files for the ServiceManager system program, which is responsible for installing and running KPA packages.

Booting KasperskyOS

1. The bootloader installed on the hardware platform or the UEFI firmware loads the KasperskyOS kernel, InitROMFS boot image, and VendorROMFS image into memory and starts the kernel.
2. The kernel finds the security module in the VendorROMFS image and loads it (as a kernel module).
3. The kernel loads the Einit system program from the InitROMFS image.
4. The Einit program starts all other system programs from the InitROMFS image and the vendor_einit system program from the VendorROMFS image.
5. The vendor_einit program from the VendorROMFS image starts all other system programs from the VendorROMFS image. This includes loading the block device drivers and file system drivers, which then allows the use of rootfs.
6. The PkgMgrEntity system program installs the ServiceManager system program from the KPA package supplied as part of rootfs.
7. The ExecMgrEntity system program starts the ServiceManager system program.
8. The ServiceManager system program installs applications from KPA packages and starts them.

KasperskyOS kernel subsystems

The diagram of interaction between applications and the KasperskyOS kernel is shown in the figure below:



Interaction of applications with the KasperskyOS kernel

In KasperskyOS, applications and drivers interact with each other and with the kernel by using the `libkos` library, which provides the interfaces for querying core endpoints (in KasperskyOS, a driver generally operates with the same level of privileges as the application). The `libkos` library queries the kernel by executing only three system calls. These system calls are `Call()`, `Recv()` and `Reply()` which implement the IPC mechanism. Kernel services are supported by kernel subsystems whose purposes are presented in the table below. Kernel subsystems interact with hardware through the hardware abstraction layer (HAL), which makes it easier to port KasperskyOS to various platforms.

Kernel subsystems and their purpose

Designation	Name	Purpose
HAL	Hardware abstraction subsystem	Basic hardware support: timers, interrupt controllers, memory management unit (MMU). This subsystem includes UART drivers and low-level means for power management.
IO	I/O manager	Registration and deallocation of hardware platform resources required for the operation of drivers, such as Interrupt ReQuest (IRQ), Memory-Mapped Input-Output (MMIO), I/O ports, and DMA buffers. If the hardware platform has an input–output memory management unit (IOMMU), this subsystem guarantees the allocation of memory used by devices.
MM	Physical memory manager	Allocation and deallocation of physical memory pages, distribution of physically contiguous page areas.
VMM	Virtual memory manager	Management of physical and virtual memory: reserving, committing, and releasing memory. Working with memory page tables for insulating the address spaces of processes.
THREAD	Thread manager	Management of threads: creating, terminating, locking, and resuming threads.
TIME	Real-time clock subsystem	Getting the time and setting the system clock. Using clocks provided by hardware.
SCHED	Scheduler	Scheduling of threads: standard threads, real-time threads, and idle threads.
SYNC	Synchronization primitive support subsystem	Implementation of basic synchronization primitives: spinlocks, mutexes, and events. The kernel supports only one primitive—futex. All other primitives are implemented based on a futex in the user space.

Designation	Name	Purpose
IPC	Interprocess communication subsystem	Implementation of a synchronous IPC mechanism based on the rendezvous principle.
KSMS	Security module interaction subsystem	This subsystem is used for working with the security module. It provides all messages relayed via IPC to the security module so that these messages can be checked.
OBJ	Object manager	Management of the general behavior of all KasperskyOS resources: tracking their life cycle and assigning unique security IDs (for details, see Resource Access Control). This subsystem is closely linked to the capability-based access control mechanism (OCap).
ROMFS	Immutable file system image startup subsystem	Operations with files from ROMFS: opening and closing, receiving a list of files and their descriptions, and receiving file characteristics (name, size).
TASK	Process management subsystem	Management of processes: creating, starting, and terminating processes. Receiving information about running processes (such as names and paths) and their exit codes.
ELF	Executable file loading subsystem	Loading executable ELF files from ROMFS into RAM, parsing headers of ELF files.
DBG	Debug support subsystem	Debugging mechanism based on GDB (GNU Debugger). The availability of this subsystem in the kernel is optional.
PM	Power manager	Power management: restart and shutdown.

IPC

Interprocess communication (IPC) is a mechanism for interaction between different processes and between a process and the KasperskyOS kernel.

IPC mechanism

Exchanging IPC messages

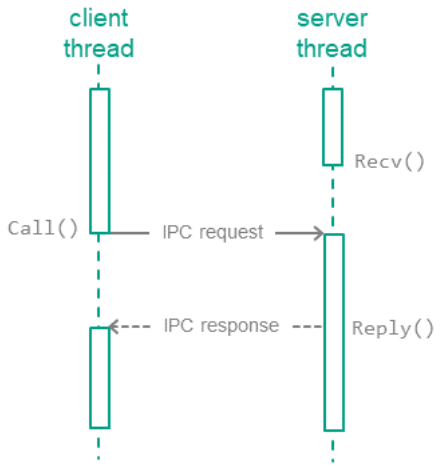
In KasperskyOS, processes interact with each other by exchanging IPC messages (*IPC request* and *IPC response*). In an interaction between processes, there are two separate roles: *client* (the process that initiates the interaction) and *server* (the process that handles the request). Additionally, a process that acts as a client in one interaction can act as a server in another.

To exchange IPC messages, the client and server use three system calls: `Call()`, `Recv()` and `Reply()` (see the figure below):

1. The client sends an IPC request to the server. To do so, one of the client's threads makes the `Call()` system call and is locked until an IPC response is received from the server.

2. The server thread that has made the `Recv()` system call waits for IPC requests. When an IPC request is received, this thread is unlocked and handles the request, then sends an IPC response by making the `Reply()` system call.

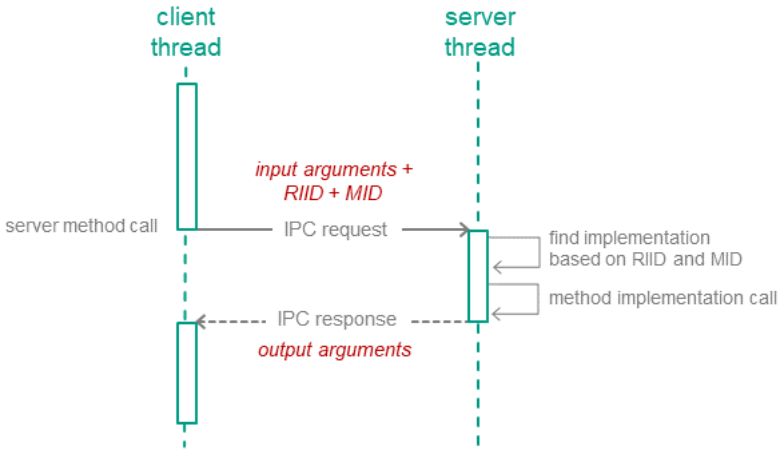
3. When an IPC response is received, the client thread is unlocked and continues execution.



Exchanging IPC messages between a client and a server

Calling methods of server endpoints

IPC requests are sent to the server when the client calls *endpoint methods* of the server (hereinafter also referred to as *interface methods*) (see the figure below). The IPC request contains input parameters for the called method, as well as the Runtime Implementation Identifier (RIID) and the called method ID (MID). Upon receiving a request, the server uses these identifiers to find the method's implementation. The server calls the method's implementation while passing in the input parameters from the IPC request. After handling the request, the server sends the client an IPC response that contains the output parameters of the method.



Calling a server endpoint method

IPC channels

To enable two processes to exchange IPC messages, an *IPC channel* must be established between them. An IPC channel has a client side and a server side. One process can use multiple IPC channels at the same time. A process may act as a server for some IPC channels while acting as a client for other IPC channels.

KasperskyOS has two mechanisms for creating IPC channels:

1. The static mechanism allows the parent process to create an IPC channel between child processes. Static creation of IPC channels is normally performed by the initializing program.

2. The dynamic mechanism allows already running processes to create IPC channels between each other.

IPC control

The Kaspersky Security Module is integrated into the IPC implementation mechanism. The security module is aware of the structure of IPC messages for all possible interactions because IDL, CDL and EDL descriptions are used to generate the source code of this module. This enables the security module to verify that the interactions between processes comply with the solution security policy.

The KasperskyOS kernel queries the security module each time a process sends an IPC message to another process. The security module operating scenario includes the following steps:

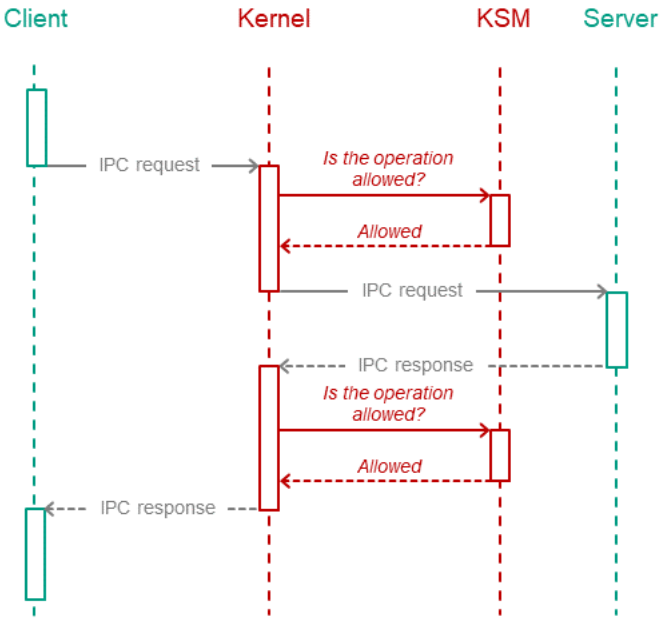
1. The security module verifies that the IPC message complies with the called method of the endpoint (the size of the IPC message is verified along with the size and location of certain structural elements).

2. If the IPC message is incorrect, the security module makes the "deny" decision and the next step of the scenario is not carried out. If the IPC message is correct, the next step of the scenario is carried out.

3. The security module checks whether the security rules allow the requested action. If allowed, the security module makes the "granted" decision. Otherwise it makes the "denied" decision.

The kernel executes the security module decision. In other words, it either delivers the IPC message to the recipient process or rejects its delivery. If delivery of an IPC message is rejected, the sender process receives an error code via the return code of the `Call()` or `Reply()` system call.

The security module checks IPC requests as well as IPC responses. The figure below depicts the controlled exchange of IPC messages between a client and a server.



Controlled exchange of IPC messages between a client and a server

Transport code for IPC

Implementation of interprocess interaction requires transport code, which is responsible for generating, sending, receiving, and processing IPC messages. However, developers of KasperskyOS-based solutions do not have to write their own transport code. Instead, you can use special tools and libraries included in the KasperskyOS SDK.

Transport code for developed components of a solution

KasperskyOS Community Edition SDK includes transport code generators that create transport code in C and C ++ from the formal specifications of solution components. The transport code includes methods and types for both clients and servers.

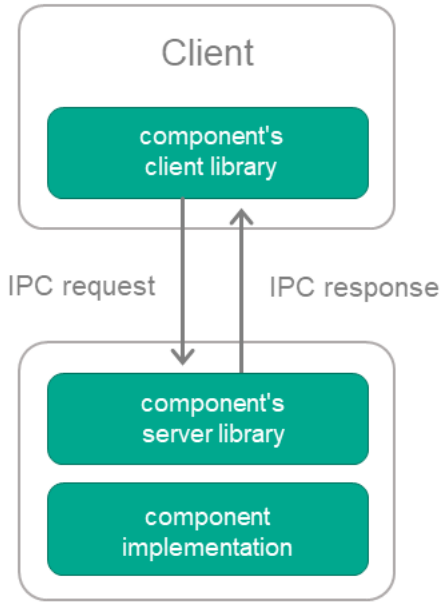
Transport code for supplied components of a solution

Most components included in the KasperskyOS SDK may be used in a solution both locally (through static linking with other components) as well as via IPC.

To use a supplied component via IPC, the KasperskyOS SDK provides the following *transport libraries*:

- *Solution component's client library*, which converts local calls into IPC requests.
- *Solution component's server library*, which converts IPC requests into local calls.

The client library is linked to the client code (the component code that will use the supplied component). The server library is linked to the implementation of the supplied component (see the figure below).



Using a supplied solution component via IPC

IPC between a process and the kernel

The IPC mechanism is used for interaction between processes and the KasperskyOS kernel. In other words, processes exchange IPC messages with the kernel. The kernel provides endpoints, and processes use those endpoints. Processes query core endpoints by calling functions of the `libkos` library (directly or via other libraries). The client transport code for interaction between a process and the kernel is included in this library.

A solution developer is not required to create IPC channels between processes and the kernel because these channels are created automatically when processes are created.

The Kaspersky Security Module controls interactions between processes and the kernel the same way it controls interactions between a process and other processes. (KasperskyOS Community Edition SDK comes with a formal specification of the solution component for the kernel that is used to generate source code of the security module.)

Resource Access Control

Types of resources

KasperskyOS has two types of resources:

- *System resources*, which are managed by the kernel. Some examples of these include processes, memory regions, and interrupts.
- *User resources*, which are managed by processes. Examples of user resources: files, input-output devices, data storage.

Handles

Both system resources and user resources are identified by *handles*. Processes (and the KasperskyOS kernel) can transfer handles to other processes. By receiving a handle, a process obtains access to the resource that is identified by this handle. In other words, the process that receives a handle can request operations to be performed with a resource by specifying its received handle in the request. The same resource can be identified by multiple handles used by different processes.

Security identifiers

The KasperskyOS kernel assigns security identifiers to system resources and user resources. A *security identifier* (SID) is a globally unique identifier of a resource (in other words, a resource can have only one SID but can have multiple handles). The Kaspersky Security Module identifies resources based on their SID.

> When transmitting an IPC message containing handles, the kernel modifies the message so that it contains SID values instead of handles when the message is checked by the security module. When the IPC message is delivered to its recipient, it will contain the handles.
> The kernel also has an SID like other resources.

Security context

Kaspersky Security System technology lets you employ security mechanisms that receive SID values as inputs. When employing these mechanisms, the Kaspersky Security Module distinguishes resources (and the KasperskyOS kernel) and binds security contexts to them. A *security context* consists of data that is associated with an SID and used by the security module to make decisions.

The contents of a security context depend on the security mechanisms being used. For example, a security context may contain the state of a resource and the levels of integrity of access subjects and/or access objects. If a security context stores the state of a resource, this lets you allow certain operations to be performed with a resource only if the resource is in a specific state, for example.

The security module can modify a security context when it makes a decision. For example, it can modify information about the state of a resource (the security module used the security context to verify that a file is in the "not in use" state and allowed the file to be opened for write access and wrote a new state called "opened for write access" into the security context of this file).

Resource access control by the KasperskyOS kernel

The KasperskyOS kernel controls access to resources by using two mutually complementary methods at the same time: executing the decisions of the Kaspersky Security Module and implementing a security mechanism based on object capabilities (OCap).

Each handle is associated with access rights to the resource identified by this handle, which means it is a *capability* in OCap terms. By receiving a handle, a process obtains the access rights to the resource that is identified by this handle. For example, these access rights may consist of read permissions, write permissions, and/or permissions to allow another process to perform operations with a resource (handle transfer permission).

Processes that use the resources provided by the kernel or other processes are referred to as *resource consumers*. When a resource consumer opens a system resource, the kernel sends the consumer the handle associated with the access rights to this resource. These access rights are assigned by the kernel. Before an operation is performed with a system resource requested by a consumer, the kernel verifies that the consumer has sufficient rights. If the consumer does not have sufficient rights, the kernel rejects the request of the consumer.

In an IPC message, a handle is sent together with its permissions mask. The *handle permissions mask* is a value whose bits are interpreted as access rights to the resource identified by the handle. A resource consumer can find out their access rights to a system resource from the handle permissions mask of this resource. The kernel uses the handle permissions mask to verify that the consumer is allowed to request the operations to be performed with the system resource.

The security module can verify the permissions masks of handles and use these verifications to either allow or deny interactions between different processes and between processes and the kernel when such interactions are related to resource access.

The kernel prohibits the expansion of access rights when handles are transferred among processes (when a handle is transferred, access rights can only be restricted).

Resource access control by resource providers

Processes that control user resources and access to those resources for other processes are referred to as *resource providers*. For example, drivers are resource providers. Resource providers control access to resources by using two mutually complementary methods: executing the decisions of the Kaspersky Security Module and using the OCap mechanism that is provided by the KasperskyOS kernel.

If a resource is queried by its name (for example, to open it), the security module cannot be used to control access to the resource without the involvement of the resource provider. This is because the security module identifies a resource by its SID, not by its name. In such cases, the resource provider finds the resource handle based on the resource name and forwards this handle (together with other data, such as the required state of the resource) to the security module via the security interface (the security module receives the SID corresponding to the transferred handle). The security module makes a decision and returns it to the resource provider. The resource provider implements the decision of the security module.

When a resource consumer opens a user resource, the resource provider sends the consumer the handle associated with the access rights to this resource. In addition, the resource provider decides which specific rights for accessing the resource will be granted to the resource consumer. Before an operation is performed with a user resource as requested by a consumer, the resource provider verifies that the consumer has sufficient rights. If the consumer does not have sufficient rights, the resource provider rejects the request of the consumer.

A resource consumer can find out their access rights to a user resource from the permissions mask of the handle of this resource. The resource provider uses the handle permissions mask to verify that the consumer is allowed to request the operations to be performed with the user resource.

Getting started with the SDK

This section tells you what you need to know to start working with KasperskyOS Community Edition SDK.

The KasperskyOS Developer's Quick Start Guide

Preparations for developing

> KasperskyOS Community Edition SDK is delivered as separate deb packages for working with the QEMU emulator and for each of the supported hardware platforms. To study the principles of KasperskyOS development, it is recommended to use the version designed for working with the QEMU emulator.

To start developing KasperskyOS solutions, complete the following steps:

1. Install a KasperskyOS Community Edition SDK version that can work with QEMU emulator and development environment, such as Microsoft Visual Studio Code with extensions for C/C++ development and CMake support, as well as KasperskyOS SDK Extension, on a computer that meets the system requirements.

   > The KasperskyOS SDK Extension for Microsoft Visual Studio Code is contained only in the SDK designed to run in the QEMU emulator and is intended only for working with this version of the SDK.

   KasperskyOS Community Edition SDK includes C/C++ compilers, source code generators, a GDB debugger, QEMU emulator, utilities, such as `binutils` and `cmake`, and other tools. Tool executables are located in `toolchain/bin`.

2. Carefully read the Overview of KasperskyOS.

3. Carefully read this section.

   When working on a project, you will need to refer to various sections to get detailed information.

Exploring the examples from KasperskyOS Community Edition SDK

The `examples` directory in KasperskyOS Community Edition SDK contains examples of KasperskyOS solution projects with a `README.md` description and comments in text files. Each example resides in a separate directory. To build an example, you need to copy it to a directory where write access is allowed (for example, to your home directory) and build it using the cmake tool. The build process will create a solution image in `<example directory>/build/einit/` directory. If the `cmake` tool is called with the `--target sim` parameter, the built image will be automatically run in QEMU. You can also build examples to be run on Raspberry Pi 4 B or OrangePi RV2. For more details, refer to the Building and running examples section.

For details on the examples, see Examples in KasperskyOS Community Edition SDK.

Some of the examples implement the security patterns that are used for KasperskyOS development.

Location of libraries and executable files of supplied solution components

The libraries (including transport libraries) of supplied solution components are located in `sysroot-*-kos/lib`, in KasperskyOS Community Edition SDK.

The executable files of supplied solution components are located in `sysroot-*-kos/bin`, in KasperskyOS Community Edition SDK.

Creating a KasperskyOS-based solution project.

You need to create a solution CMake project. To do so, you need to create a hierarchy of directories and files as described in the Using CMake from the contents of KasperskyOS Community Edition SDK section (similar to the project examples). In addition to files containing source code, a solution CMake project includes an `*.sh` build script, `CMakeLists.txt` files, `*.psl` files, and `*.idl`, `*.cdl`, and `*.edl` files.

Creating a build script

A build script can be created based on the description in the section titled Using CMake from the contents of KasperskyOS Community Edition SDK. The build target is determined by the value of the `--target` parameter when `cmake` is called. For example, a build target may be to create and run a solution image in QEMU or to create a solution image to be run on a hardware platform.

Creating build scenarios (CMakeLists.txt files)

`CMakeLists.txt` files contain build scenarios. Create one root `CMakeLists.txt` file and then one `CMakeLists.txt` file for each program included in the solution. To create `CMakeLists.txt` files, use information from Using CMake from the contents of KasperskyOS Community Edition SDK and the project examples.

`CMakeLists.txt` files contain standard CMake commands (for example, `add_executable()`, `add_library()`, and `target_link_libraries()`) as well as CMake commands that are specific to KasperskyOS solution projects.

Creating a solution security policy description (security.psl files)

The Kaspersky Security Module is generated during the solution build process based on the following components of the solution security policy description:

- Description of the security policy for system programs included in the InitROMFS boot image. This description is provided as part of the SDK.
- Security policy description for system programs included in the additional VendorROMFS image. This description is generated while building this image.
- Security policy description for applications. This description is created by the SDK user. For more details, refer to Developing security policies.

The `security.psl` file contains part of the solution security policy description and refers to the files containing its other parts. For example, the part of the solution security policy description that manages interactions between programs and the KasperskyOS kernel is placed into `core.psl`.

In the `security.psl` file, you can include declarations for performing a security audit.

During the early stages of development, you can use a solution security policy stub that allows all interactions (see Example descriptions of basic security policies for KasperskyOS-based solutions).

Creating formal specifications of solution components (*.idl, *.cdl, and *.edl files)

For each program in the solution, create a formal specification that describes the program in terms of its interaction with other programs. For example, the formal specification defines whether the program provides endpoints to other programs, which endpoints are provided, which interfaces they have, and whether the program can query the Kaspersky Security Module via the security interface.

The `*.edl` file is mandatory, while `*.cdl` and `*.idl` are optional. If the program does not provide endpoints to other programs in the solution, its formal specification may consist of one `*.edl` file.

The formal specification files for supplied solution components are located in `sysroot-*-kos/include/kl`, in KasperskyOS Community Edition SDK. There is also a formal specification of the KasperskyOS kernel, whose files are located in `sysroot-*-kos/include/kl/core`, in KasperskyOS Community Edition SDK. The solution component executable is accompanied by the full set of formal specification files. The libraries for creating a solution component are accompanied by `*.cdl` and `*.idl` files.

You need to create formal specifications for programs being developed as part of the project. You can use a CMake command to create `*.edl`. If the program executable is linked to libraries provided in KasperskyOS Community Edition SDK, include the `*.cdl` and `*.idl` files accompanying these libraries into the formal specification of the program.

Creating IPC channels

IPC channels can be created statically or dynamically. See Creating IPC channels.

Creating and using transport code

To create transport code in C, you can use the `nk_build_*dl_files()` CMake commands (see nk library). These CMake commands create the `*.edl.h`, `*.cdl.h`, and `*.idl.h` files, which contain the transport code. These files need to be included in the source code of both clients and servers. The specific files to include depend on which transport code elements are required. For example, it may suffice to include `*.idl.h` into the client code and `*.edl.h` into the server code. However, a client may also require the constants defined in `*.edl.h`. You can use the `NK_FLAGS` parameter of the CMake commands `nk_build_*dl_files()` to define the transport code generation parameters that can also modify the contents of the `*.edl.h`, `*.cdl.h`, and `*.idl.h` files.

Examples of using transport code elements written in C are provided in Initializing IPC transport for interprocess communication and managing IPC request processing (transport-kos.h, transport-kos-dispatch.h) and Initializing IPC transport for querying the security module (transport-kos-security.h).

For details about creating and using C++ transport code, see Transport code in C++ and Generating transport code in C++.

If a solution component is supplied with transport libraries, you do not need to create transport code.

Querying the Kaspersky Security Module

A security interface must be defined in the formal specification of the program. An example of code for querying the Kaspersky Security Module is provided in Initializing IPC transport for querying the security module (transport-kos-security.h).

Working with IPC messages

An IPC message contains a constant part and an (optional) arena. The constant part is a structure that serves as an element of transport code. Operations on the constant part of IPC messages consist of writing and reading fields of the structure. You must use an API to work with the IPC message arena (see Working with an IPC message arena).

Using dynamic libraries

You can use dynamic libraries (`*.so`). See Using dynamic libraries.

Working with file systems and network stack

KasperskyOS Community Edition SDK is supplied with executable files and libraries containing implementations of file systems and the network stack. Functionality is available to programs via POSIX functions and other functions of the standard C library.

See File systems and network.

Working with KPA packages

A KPA package is essentially packaging for a program intended for installation in a KasperskyOS-based solution.

Scenario for working with KPA packages:

- During the build, applications must be packed into KPA packages using the CMake library platform/kpa.
- Applications packed into KPA packages are run by using the ServiceManager system program.
- All examples in the KasperskyOS Community Edition SDK use the KPA package mechanism.

KasperskyOS Community Edition SDK includes the following components and tools used to work with KPA packages:

- Tools for managing KPA packages let you build a KPA package from program source files in the system where the KasperskyOS Community Edition SDK is installed and then install the KPA package in a built KasperskyOS-based solution image.
- PackageManager component lets you install KPA packages in an operational KasperskyOS-based solution, remove KPA packages, and receive information about them.

See Working with KPA packages.

KasperskyOS APIs

KasperskyOS provides the following APIs:

- POSIX (with limitations and specific features of implementation) and other APIs of the standard C library:
  - Main APIs (header files in the `sysroot-*-kos/include/strict` and `sysroot-*-kos/include/sys` directories from KasperskyOS Community Edition SDK)
  - Extended APIs (header files in `sysroot-*-kos/include` from KasperskyOS Community Edition SDK)

  The header files of extended APIs include the header files of the main APIs (the `include` directive is used to include header files located in `sysroot-*-kos/include/strict`, in KasperskyOS Community Edition SDK).
- Native APIs:
  - Top-level APIs (header files in `sysroot-*-kos/include/kos`, in KasperskyOS Community Edition SDK)
  - Low-level APIs (header files in `sysroot-*-kos/include/coresrv`, in KasperskyOS Community Edition SDK)

Components for application development

KasperskyOS Community Edition SDK includes the following:

- The LogRR component, which provides a system for logging information about the operation of other programs.
- The TLS Terminator component, which ensures secure data transmission over the network between the client and the server using the Transport Layer Security (TLS) protocol.

Debugging programs

To debug programs, use the GDB debugger from KasperskyOS Community Edition SDK. Debugging can be performed in QEMU or on the hardware platform.

If a process terminates unexpectedly, the KasperskyOS kernel prints stack backtrace data for the thread in which the unhandled exception occurred. This data lets you determine the sequence of function calls that led to the unexpected termination of the process.

A handle sanitizer lets you detect the use of invalid handles (closed and uncreated) and leaked handles (the presence of unclosed handles that are no longer required).

See Debugging programs in a KasperskyOS-based solution.

Drivers for hardware platforms

The SDK includes drivers for supported hardware platforms.

Developing drivers

In KasperskyOS, a driver may be a bus driver and/or a client driver, and it may be local or distributed among processes. For driver development, KasperskyOS Community Edition SDK provides the `kdf` library (header files in `sysroot-*-kos/include/kdf`, in KasperskyOS Community Edition SDK).

See Developing drivers for KasperskyOS.

Using a glossary

The glossary defines terms and abbreviations in alphabetical order. Each term is accompanied by a definition and links to the sections related to the term.

Using a Docker container

To install and use KasperskyOS Community Edition SDK, you can use a Docker container in which an image of one of the supported operating systems is deployed.

To use a Docker container for installing KasperskyOS Community Edition SDK:

1. Make sure that the Docker software is installed and running.
2. To download the official Docker image of the Ubuntu GNU/Linux 24.04 (Noble Numbat) operating system from the public Docker Hub repository, run the following command:

   `docker pull ubuntu:24.04`
3. To run the image, run the following command:

   `docker run --net=host --user root --privileged -it ubuntu:24.04 bash`
4. Open a second terminal and get the ID of the container with the running image by using the `docker ps` command.
5. In the same terminal, copy the deb package for installing the KasperskyOS Community Edition SDK into the container with the obtained ID by running the following command:

   `docker cp KasperskyOS-Community-Edition-<version>.deb <container_id>:/opt`
6. In the first terminal where the Docker container was started, install the KasperskyOS Community Edition SDK.

Installation and removal

Installation

KasperskyOS Community Edition SDK is distributed as a deb package. It is recommended to use the `apt` package installer to install KasperskyOS Community Edition SDK.

To deploy the package using `apt`, run the following command:

```
$ sudo apt update && sudo apt install ./KasperskyOS-Community-Edition-<version>.deb
```

The package will be installed in the `/opt/KasperskyOS-Community-Edition-<platform>-<version>` directory.

Setting values of environment variables

To conveniently work with tools provided in the KasperskyOS Community Edition SDK, the path to the executable files of these tools in `/opt/KasperskyOS-Community-Edition-<platform>-<version>/toolchain/bin` must be added to the `PATH` environment variable. To do so, the following commands must be executed:

```
$ cd /opt/KasperskyOS-Community-Edition-<platform>-<version>/common/
$ source set_env.sh
```

In addition to changing the `PATH` environment variable, the script also defines the `KOSCEVER` and `KOSCEDIR` environment variables that contain the version and absolute path to the KasperskyOS Community Edition SDK, respectively. Use of these environment variables allows to determine the SDK installation path, and to verify that the solution version matches the SDK version.

If the environment variables were successfully defined, the `printenv | grep KOSCE` command will print the paths contained in the variables.

> Environment variables are recognized only in the current terminal. When using a different terminal, the variables must be defined again.

Removal

To remove KasperskyOS Community Edition SDK, run the following command:

```
$ sudo apt remove --purge kasperskyos-community-edition-<platform>
```

After running this command, all installed files in the `/opt/KasperskyOS-Community-Edition-<platform>-<version>` directory will be deleted.

Configuring the development environment

This section provides brief instructions on configuring the development environment and adding the header files included in KasperskyOS Community Edition SDK to a development project.

Configuring the code editor

Before getting started, you should do the following to simplify your development of KasperskyOS-based solutions:

- Install code editor extensions and plugins for your programming language (C, C++ and/or Rust).
- Add the header files included in KasperskyOS Community Edition SDK to the development project.

  The header files are located in the following directory: `/opt/KasperskyOS-Community-Edition-<platform>-<version>/sysroot-*-kos/include`.

Example of how to configure Visual Studio Code

KasperskyOS Community Edition SDK includes the KasperskyOS SDK Extension for Visual Studio Code, which integrates KasperskyOS Community Edition SDK with the source code editor known as Visual Studio Code.

> The KasperskyOS SDK Extension for Microsoft Visual Studio Code is provided only in the QEMU emulator SDK and intended only for this SDK version.

Instead of installing the extension, you can also manually add the header files included in KasperskyOS Community Edition SDK to the development project.

To more conveniently navigate the project code, including the system API:

1. Create a new workspace or open an existing workspace in Visual Studio Code.

   A workspace can be opened implicitly by using the `File` > `Open folder` menu options.
2. Make sure the C/C++ for Visual Studio Code extension is installed.
3. In the `View` menu, select the `Command Palette` item.
4. Select the `C/C++: Edit Configurations (UI)` item.
5. In the `Include path` field, add the path `/opt/KasperskyOS-Community-Edition-<platform>-<version>/sysroot-*-kos/include`.
6. Close the `C/C++ Configurations` window.

KasperskyOS SDK Extension for Visual Studio Code

The KasperskyOS SDK Extension for Visual Studio Code (hereinafter also referred to as the KasperskyOS SDK Extension) integrates the KasperskyOS Community Edition SDK with the source code editor known as Visual Studio Code.

> The KasperskyOS SDK Extension for Microsoft Visual Studio Code is contained only in the deb package to be run in the QEMU emulator.

The extension provides access to some operating scenarios of KasperskyOS Community Edition SDK from the graphical user interface of Visual Studio Code. For example, it lets you do the following:

- Create a project based on a template.
- Start a project build.
- Test programs.

To ensure access to these and other features, the extension provides the following:

- Commands in the command palette
- Buttons in the lower panel
- Controls in the list of devices and emulators

The extension also provides a set of configurable parameters.

Extension functions

The extension automatically detects a KasperskyOS project in the open workspace and starts. The detection parameter is the presence of the `.vscode/kos_project.json` file or `einit` directory in the workspace. If the extension has not automatically activated in the directory but you are certain that it is a KasperskyOS project, run the command `KOS: Activate extension in this directory`. This command activates the extension and creates an empty `.vscode/kos_project.json` file in the project directory. To cancel manual activation, delete this file.

After startup, the extension adds the buttons listed in the article titled Buttons in the lower panel to the lower panel of the Visual Studio Code editor.

Functions that provide buttons can also be called from the command line of the code editor, which can be opened by pressing **F1**. All commands have the `KOS:` prefix.

Working with a basic KasperskyOS image

The extension lets you avoid having to completely build a KasperskyOS-based solution after making changes to application code. Instead, you can use the following scenario:

1. Running a basic KasperskyOS-based solution image, included in the SDK, in the QEMU emulator. The basic image contains all the system programs required to run and debug programs packaged in KPA packages.
2. Build an application and package it into a KPA package. You must use the `CMake` commands of the kpa library to pack a program into a KPA package.
3. Install a program from a KPA package into the solution image that was run at step 1 and start it under KasperskyOS.
4. Make changes to the application code.
5. Repeat steps 2–4.
6. With the extension, you can also debug a program that is running in a basic KasperskyOS image. For more details, refer to Debugging programs that are part of a KPA package.

Working with a solution security policy description

The extension lets you do the following:

- Use PSL syntax highlighting.
- Diagnose errors when writing a solution security policy.
- Use name autocompletion and code navigation (`"Go to Definition"`) when creating PSL files.

> The functionality for working with PSL is not supported if Ginger templating is used in the PSL file.

Starting solution security policy tests

The extension lets you run solution security policy tests for a project if they were created based on the solution security policy description.

To build and run solution security policy tests:

1. Make sure that the solution security policy description contains the solution security policy tests.
2. Make sure that the `CMake` commands for building solution security policy tests have been added to one of the `CMakeLists.txt` files of the project.

3. Make sure that the KasperskyOS SDK Extension is installed and active.

4. Build all targets by clicking the button in the lower panel.

5. Click the build target selection button all and select the target named `kos-qemu-image-PalTest<N>-sim` (N refers to the PSL file index within the list of PSL files containing solution security policy tests).

6. Click the start button for the selected target in the lower panel ▷.

It is recommended to either write the solution security policy tests in a separate project or remove the CMake commands for creating an image from the solution, because the code for creating the tests already contains the code responsible for creating an image. For more details, refer to the pal_tests example in the SDK.

Extension usage scenarios

Installing and removing the extension

Installing dependencies

Extension dependencies are automatically installed from the Visual Studio Code Extension Marketplace. If your computer has access to the Visual Studio Code Extension Marketplace, you can skip the steps for setting dependencies below and instead proceed to the "Installing the extension" section.

Otherwise, you will need to manually set the dependencies for the extension. To set the dependencies of the KasperskyOS SDK Extension for Visual Studio Code:

1. On another computer that can access the Visual Studio Code Extension Marketplace, open the address https://marketplace.visualstudio.com/vscode in a browser.

2. Find and download the relevant extensions:

   - `cpptools` is required for working with the debugger.

   - `clangd` is required for static analysis of source code.

3. Copy the received VSIX files to the computer where you need to install the KasperskyOS SDK Extension.

4. Open the Visual Studio Code development environment.

5. Press **F1**.

6. Run the `Extensions: Install from VSIX...` command in the command line that opens in the upper part of the window.

7. Install the extensions that you downloaded at step 2.

Installing the extension

To install KasperskyOS SDK Extension, do the following:

1. Open the Visual Studio Code development environment.
2. Press **F1**.
3. Run the `Extensions: Install from VSIX...` command in the command line that opens in the upper part of the window.
4. Select the `kos-extension-<version>.vsix` extension file located in the directory `/opt/KasperskyOS-Community-Edition-<platform>-<version>/dev_tools/ide_integration/vscode/`.

   During installation of the KasperskyOS SDK Extension, its dependencies are automatically set.

Removing the extension

To remove the extension, press the key combination **CTRL**+**SHIFT**+**X**, find the extension in the list, and select the **Uninstall** command from the context menu of the extension.

Working with a Docker container in Visual Studio Code

You should first start the Docker container and download the KasperskyOS Community Edition SDK.

To work with the KasperskyOS Community Edition SDK inside a Docker container using the KasperskyOS SDK Extension:

1. In the terminal where the Docker container was started, install the KasperskyOS Community Edition SDK:

   ```
   sudo apt update && sudo apt install /opt/KasperskyOS-Community-Edition-<version>.deb
   ```

2. In the same terminal, set the values of the environment variables.
3. Install the `Dev Containers` extension on the host system for Visual Studio Code.
4. In Visual Studio Code, open the Remote Explorer tab.
5. In this tab, select the necessary container and connect to it.
6. Install the KasperskyOS SDK Extension, which lets you work with the Docker container in Visual Studio Code.

Updating the extension

The version of the KasperskyOS SDK Extension must match the version of KasperskyOS Community Edition SDK that you are using.

To update the extension after installing a new version of KasperskyOS Community Edition SDK:

1. Make sure that the Auto Update check box is selected in the extension settings (this check box is selected by default).
2. Click the button containing the name of the current SDK in the lower panel or run the command `KOS: Change SDK for the current project` in the command palette.
3. Select a new path to KasperskyOS Community Edition SDK from the list of SDKs detected on your computer.

   The extension will be automatically reinstalled using the VSIX installation file from the new SDK.

Using multiple SDKs

You can use the command `KOS: Change SDK for the current project` to switch to another KasperskyOS SDK when using multiple SDKs.

Creating a project in Visual Studio Code

To create a project using the KasperskyOS SDK Extension:

1. Open the VS Code command palette and run the command `KOS: New project`.
2. In the displayed list of paths to installed KasperskyOS SDKs, select the path to the SDK.
3. In the displayed list of templates and project examples available in the SDK, select the template or example that you want to use as the basis for creating a project.
4. If you have selected a template, specify the template settings.
5. Specify the directory for the project being created.

   The project creation process will start. At the end of the process, a notification window will appear.
6. If you want to open the created project, click one of the following buttons in the notification window:

   - **Open in current window** to open the project in the current Visual Studio Code window.
   - **Open in new window** to open the project in a new Visual Studio Code window.

   If you close or ignore the notification, you can manually open the created project later by selecting the project directory.

Starting a basic KasperskyOS-based solution image

The basic KasperskyOS-based solution image is included in the SDK and contains all the system programs required to run and debug programs packaged in KPA packages.

To start a basic KasperskyOS-based solution image from Visual Studio Code:

1. Make sure that the KasperskyOS SDK Extension is installed and active.
2. Click the button for selecting a device or emulator (displays the name of the current device or `No device`) in the lower panel, or press the `F1` key and run the command `KOS: Select Device`.
3. Select **Create new emulator**.
4. Select **kos_base-dev**.
5. In the opened field, enter the additional QEMU flags and press **Enter**, or immediately press **Enter** to use the default flags.
6. In the opened field, enter the name of the new emulator and press **Enter**, or immediately press **Enter** to use the automatically generated name.
7. Wait until the previous command completes.
8. In the notification about the new emulator being created, click the **Start** button.
9. Wait for the emulator to finish loading.

   After the emulator has finished loading, the output **Kaspersky Debug Service ready** will appear on the `TERMINAL` tab.

Building programs in Visual Studio Code

To build a program in Visual Studio Code:

1. Open the program project directory in Visual Studio Code.
2. Make sure that the program has been packed into a KPA package.

   You must use the `CMake` commands of the kpa library to pack a program into a KPA package.

   > Important! When creating a startup configuration using the `CMake` command `add_kpa_component()`, you must specify the `kl.Kds` value in the `EIID` field.

3. Make sure that the project was correctly identified based on the appearance of extension buttons in the lower panel. If the buttons do not appear, manually activate the solution by running the command `KOS: Activate extension in this directory`.
4. Select the build architecture by clicking the build architecture selection button in the lower panel (the button shows the current architecture, for example: `aarch64-kos`).
5. Build all targets by clicking the ⚙ button in the lower panel.

Starting programs in a basic solution image

To start a program in a basic solution image:

1. Open the program project directory in Visual Studio Code.
2. Make sure that the project was correctly identified based on the appearance of additional extension buttons in the lower panel. If the buttons do not appear, manually activate the solution by running the command **KOS: Activate extension in this directory**.
3. Make sure that the basic solution image is started according to the instructions provided in the Starting a basic KasperskyOS-based solution image section.
4. Click the button for selecting the device or QEMU emulator No device in the lower panel.
5. Select the previously started basic solution image.
6. Build the program according to the instructions provided in the Building a program in Visual Studio Code section.
7. Make sure that the program has been packed into a KPA package.

   You must use the `CMake` commands of the kpa library to pack a program into a KPA package.

   > Important! When creating a startup configuration using the `CMake` command `add_kpa_component()`, you must specify the `kl.Kds` value in the `EIID` field.

8. Click the button in the lower panel that displays the name of the build target (for example: `all`). In the drop-down list, select the built program KPA package that is signed as `[application]` in the list.
9. Click the start button ▷ in the lower panel. The program will be installed to the selected device or an emulator and will automatically start.

   > When you click this button, the program starts with the default parameters (without arguments and environment variables). If you need to specify the startup parameters, you can start the program using the launch.json startup configuration.

Starting programs using a launch.json startup configuration in Visual Studio Code

To start KasperskyOS Community Edition SDK using a launch.json startup configuration:

1. Open the program project directory in Visual Studio Code.
2. On the side bar of Visual Studio Code, click **Run and debug** > **create a launch.json file** and then select **KasperskyOS Debugger**.

   This will create a debug configuration file named `launch.json`.

The created `.vscode/launch.json` file contains sets of configurations with the following names:

- `(kos/gdb) Attach to process` – configuration for connecting to the running process.
- `(kos/gdb) Launch & debug executable` – configuration for running the executable file and connecting the debugger to the running process.
- `(kos/gdb) Launch & debug application` – configuration for running the program and connecting the debugger to the running process.
- `(kos/gdb) Launch & debug qemu` – configuration for running an emulator in debug mode.
- `(kos) Launch executable` – configuration for running the executable file.
- `(kos) Launch application` – configuration for running the program.
- `(kos) Launch qemu` – configuration for running the emulator.

3. In the created `.vscode/launch.json` file, find the `(kos) Launch application` section containing the application startup parameters:

```
.vscode/launch.json (fragment)
```

```
{
    "name": "(kos) Launch application",
    "type": "kosgdb",
    "request": "launch-app",
    "app": "app.id",
    "noDebug": "true"
}
```

4. Add the necessary parameters and their values to the found section.

   You can get a list of the available parameters in the tips that appear when you edit the file.

5. Save the changes to the `.vscode/launch.json` file.

6. Start the program:

   a. Return to the **Run and debug** tab.

   b. In the drop-down list in the upper part of the tab, select the `(kos) Launch application` startup configuration that you edited.

   c. Click the ▷ button next to the drop-down list, or press the **F5** key.

> Before starting the application, you may need to install the application on the device by using the `KOS: Install package` command or by setting the parameter `preLaunchTask = "true"` in the `.vscode/launch.json` file.

Viewing event logs

The KasperskyOS SDK Extension allows you to display the following types of event logs in the integrated terminal of the Visual Studio Code editor:

- System event log of KasperskyOS – for running emulators.
- KasperskyOS emulator event log – for running emulators (includes emulator messages and the system event log of KasperskyOS running in the emulator).

Viewing the system event log of KasperskyOS

To view the system event log of KasperskyOS:

1. Click the device selection button in the lower panel of the editor (showing the name of the current emulator) or run the KOS: Select Device `command` in the command palette.

   A list of available emulators is displayed.

2. Click the button in the line corresponding to the relevant emulator.

Viewing the event log of the KasperskyOS emulator

When the KasperskyOS emulator is started from the Visual Studio Code editor, the emulator event log is displayed in the integrated terminal of the editor. If you close the terminal or restart Visual Studio Code after starting the emulator, the log can be displayed in the terminal again.

To view the emulator event log:

1. Click the device selection button in the lower panel of the editor (showing the name of the current emulator) or run the KOS: Select Device `command` in the command palette.

   A list of available emulators is displayed.

2. Click the button in the line corresponding to the relevant emulator.

The number of available log lines is specified by the `extension parameter` Show Last Logs Num (`500` by default). Important note: to display the log, the extension must be connected to the KDS server, which becomes available after KasperskyOS Community Edition SDK finishes starting up.

Debugging a program that is part of a KPA package

Unlike debugging programs in a KasperskyOS-based solution, debugging a program in a KPA package means you don't have to rebuild the entire KasperskyOS-based solution after updating the application code.

The scenario for debugging a program that is part of a KPA package looks as follows:

1. Install the KasperskyOS SDK Extension.
2. Build an application and package it into a KPA package. You must use the `CMake` commands of the kpa library to pack a program into a KPA package.
3. Run a basic KasperskyOS image from KasperskyOS Community Edition SDK with a GDB server built into the kernel.
4. Connect the debugger to the GDB server of the kernel. The debugger can be connected at program startup or while the program is running. Details:
   - Connecting a debugger at program startup.
   - Connecting a debugger while a program is running.
5. Debugging a program using the Visual Studio Code graphical user interface. For more details, refer to the Visual Studio Code documentation: https://code.visualstudio.com/docs/editor/debugging.

Connecting a debugger when starting a program in a basic image

To connect a debugger to the GDB server of the kernel while starting a program in QEMU using the Visual Studio Code extension:

1. Set a breakpoint in the program source code before the fragment to be debugged.
2. Build the program after preparing it for debugging. You must use the `CMake` commands of the kpa library to pack a program into a KPA package. You must use the `CMake` commands of the kpa library to pack a program into a KPA package.
3. Run a basic KasperskyOS image from KasperskyOS Community Edition SDK.
4. Wait for the basic image to finish loading and install the program by running the command `KOS: Install package`.
5. On the side bar of Visual Studio Code, click **Run and debug** > **create a launch.json file** and then select **KasperskyOS Debugger**.

   This will create a debug configuration file named `launch.json`.
6. In the `launch.json` file, in the field of the configuration named `(kos/gdb) Launch & debug application`, specify the path to the binary file of your program resulting from the build, and the program name. In the `eiid` field, you need to specify the `kl.Kds` value.
7. Start debugging by clicking the `(kos/gdb) Launch & debug application` button on the lower pane.
8. In the drop-down list, select the configuration named `(kos/gdb) Launch & debug application`.
9. The debug console will show a message stating that the program is ready for debugging. Click **Continue**, and program execution will be stopped at the breakpoint that you selected at step 1.

Connecting a debugger while a program is running from a basic image

To connect a debugger to the GDB server of the kernel while the program is running in QEMU using the Visual Studio Code extension:

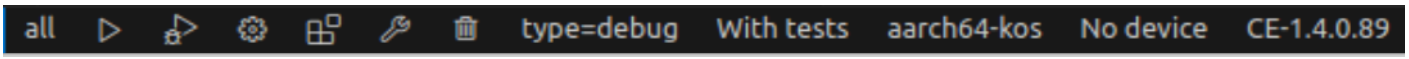
1. Add an infinite loop to the `main` function of the program. This enables the debugger to connect to the running program.
2. Set a breakpoint in the program source code before the fragment to be debugged.
3. Build the program after preparing it for debugging. You must use the `CMake` commands of the kpa library to pack a program into a KPA package.
4. Run a basic KasperskyOS image from KasperskyOS Community Edition SDK.
5. On the side bar of Visual Studio Code, click **Run and debug** > **create a launch.json file** and then select **KasperskyOS Debugger**.

   This will create a debug configuration file named `launch.json`.
6. In the `launch.json` file, in the field of the configuration named `(kos/gdb) Attach to process`, specify the path to the binary file of your program resulting from the build.
7. Start the program in the basic image.
8. Start debugging by clicking the `(kos/gdb) Attach to process` button on the lower pane.
9. In the drop-down list, select the configuration named `(kos/gdb) Attach to process` and select the name of the program to debug.
10. The debug console will show a message stating that the program is ready for debugging. Click **Continue**, and program execution will be stopped at the breakpoint that you selected at step 2.

Extension reference information

Buttons in the lower panel

The activated extension adds buttons to the lower panel of the Visual Studio Code editor:



Extension buttons in the lower panel of Visual Studio Code

These buttons are listed in the table below in the order of their appearance on the panel, from left to right. Please take the following into consideration when using this table:

- Icons are displayed on some of the buttons (shown in the Icon column). Buttons without icons are used to control the current state (for example, to select the build target or utilized architecture). Instead of icons, these buttons display the current selection (for example, the name of the build target or architecture).
- The device/emulator selection button does not have an icon by default (the text `No device` is displayed). However, after selection, the button displays both the device/emulator name and the icon indicating the status of the device/emulator.
- All buttons have pop-up tooltips that are displayed when you hover your mouse cursor over the button (listed in the Tooltip column).
- Some of the buttons are duplicated by commands in the command palette (indicated in the Duplicate command column).

Extension buttons in the lower panel of Visual Studio Code

Icon	Tooltip	Button description	Duplicate command
No	**Selected target**	Displays the name of the selected build target (for example: `all`). When you click the button, a list of available build targets where you can select a new target is displayed.	`KOS: Select build target`
▷	**Run the selected application**	Starts the selected build target. The `KOS: Select device` command selection has no effect on this command.	`KOS: Run project in QEMU`
	Debug selected target	Starts debugging the currently selected target.	`KOS: Start debugging`
	Build all targets	Builds the `all` build target for the currently selected architecture.	`KOS: Build project`
	Build the selected target	Starts building the currently selected target.	No
	Run CMake	Runs CMake to configure the current project's build directory for the selected build architecture.	`KOS: Run cmake`
	Clean	Cleans the build directory for the current project.	`KOS: Clean build directory`
No	**Build features**	Displays the selected build options (for example: `type=debug`). When you click the button, a list of available build options where you can select one or more options is displayed.	No
No	**Build with tests**	If the project is being built with tests, the button displays the text `With tests`. Otherwise it shows `Without tests`. When you click the button, a list in which you can enable or disable the build with tests by selecting `With tests` or `Without tests` is displayed.	No
No	**Architecture**	Displays the name of the selected architecture (for example: `aarch64-kos`). When you click the button, a list of available architectures where you can select a new architecture is displayed.	No
No 🔴 🟢 ?	**KasperskyOS device**	Displays the name and status of the selected emulator. The following icons are used to indicate the status: • 🔴 – inactive status • 🟢 – active status • ? – unknown status If no emulator is selected, the button displays the text `No device` (in this case, the status indicator icon is not displayed). When you click the button, a list of available emulators in which you can select an emulator, create a new emulator, stop an emulator, and view the event logs is displayed.	`KOS: Select device`
No	**KasperskyOS SDK**	Displays the short name of the KasperskyOS SDK that you are using. When you click the button, a list of paths to the installed SDKs where you can select a new SDK is displayed. Use this button to specify a new path to the KasperskyOS SDK and update the KasperskyOS SDK Extension after updating or reinstalling the KasperskyOS SDK. You can also use this button to switch to another KasperskyOS SDK when multiple SDKs are installed.	`KOS: Change SDK for the current project`

Commands in the command palette

The extension adds the following commands to the command palette in the Visual Studio Code editor:

`KOS: Activate extension in this directory`

Activates the KasperskyOS SDK Extension and creates an empty `.vscode/kos_project.json` file in the working directory.

`KOS: Add SDK to the user's SDK list`

Prompts to choose a new KasperskyOS SDK path in the file open dialog and adds it to the SDK selection list.

`KOS: Build project`

Builds the `all` build target for the currently selected architecture. You can also execute this command by clicking the gear button in the status bar.

`KOS: Change SDK for the current project`

Prompts to choose a new KasperskyOS SDK path. Use this command to update the KasperskyOS SDK Extension after updating or reinstalling the KasperskyOS SDK. You can also use this command to switch to another KasperskyOS SDK when multiple SDKs are installed. If you were using the kosctl tool in the integrated terminal prior to executing this command, reopen the terminal to refresh the path to the kosctl executable.

`KOS: Clean build directory`

Cleans the build directory for the current project. You can also execute this command by clicking the trash button in the status bar.

`KOS: Configure clang-format`

Sets the clang VS Code extension as the default formatter for C/C++ languages.

`KOS: Configure clang-tidy`

Configures the `clang-tidy` tool for C/C++ code static analysis. Requires the `cpptools` VS Code extension to be installed. Updates the path to the `clang-tidy` binary (if the binary exists in the KasperskyOS SDK directory) and enables `clang-tidy` for the workspace.

`KOS: Configure clangd`

Configures the Clangd language server for C/C++ development. Requires the `clangd` VS Code extension to be installed. Disables the IntelliSense engine provided by the Microsoft's C/C++ extension to avoid conflicts with Clangd. Updates Clangd settings based on the KasperskyOS SDK path and build directory. In particular, it updates the path to the Clangd binary (if the binary exists in the SDK directory).

`KOS: Disable extension in this directory`

Disables the KasperskyOS SDK Extension in the current directory and deletes the `.vscode/kos_project.json` file.

`KOS: Install package`

Installs the specified KPA package in the currently selected emulator or physical device. Prompts to pick a KPA file in the file open dialog.

`KOS: Mark the current file as a manifest`

Marks the file opened in the editor as a manifest. Registers the `kos://` URI scheme as a manifest schema if it is not yet registered.

`KOS: New project`

Creates a new KasperskyOS SDK project based on a selected template or example project supplied with SDK.

`KOS: Open documentation`

Opens the KasperskyOS SDK documentation in the default system browser.

`KOS: Open documentation for current project`

Opens the current project's documentation (if any). Examples, shipped with the KasperskyOS SDK have their own documentation.

`KOS: Reset documentation suggestions`

Resets the documentation suggestions.

`KOS: Restart NK language server`

Restarts the KasperskyOS NK Language Server.

`KOS: Run cmake`

Runs CMake to configure the current project's build directory for the selected build architecture. You can also execute this command by clicking the wrench button in the status bar.

`KOS: Run project in QEMU`

Starts the `sim` build target that runs the project in an emulator associated with this target. The `KOS: Select device` command selection has no effect on this command.

`KOS: Select build target`

Prompts to choose a build target for the current project. You can also execute this command by clicking the previously selected target name in the status bar.

`KOS: Select device`

Prompts to choose an emulator or a physical device to connect to, or create a new emulator. You can also execute this command by clicking the device name (or `No device` if no device is selected) in the status bar.

`KOS: Set build directory for the current project`

Prompts to choose the current project's build directory in the directory open dialog.

`KOS: Show resource monitor`

Opens the resource monitoring window for the currently selected emulator or physical device. You can also execute this command by clicking the resource monitor button in the device selection list (the list opened by the `KOS: Select device` command).

`KOS: Start application`

Starts the specified program in the currently selected emulator or physical device. Prompts to choose an identifier of a program to start.

`KOS: Start debugging`

Starts debugging the currently selected target. You can also execute this command by clicking the debug button in the status bar.

`KOS: Uninstall package`

Uninstalls the specified KPA package from the currently selected emulator or physical device. Prompts to choose an identifier of a package to uninstall.

`KOS: Unmark the current file as a manifest`

Marks the file opened in the editor as not a manifest.

Controls in the list of devices and emulators

When you run the command `KOS: Select device` or click the KasperskyOS device **button**, a list of available devices and emulators is displayed.

In the lines of this list, buttons are available for interaction with the corresponding device or emulator.

Extension buttons in the list of devices and emulators

Icon	Tooltip	Button description
□	**Stop**	Stops the emulator.
🕮	**Open system logger journal**	Opens the KasperskyOS system event log.
📋	**Open emulator console output**	Opens the emulator event log.
×	**Remove**	Removes the emulator.

In addition to buttons, each line of the list may display an indicator icon showing the state of the connection to a device or emulator.

Indicator icon states:

Icon	Connection state
🔴	Connection is inactive.
🟢	Connection is inactive.
?	Status unknown

You can select a device or emulator by clicking the corresponding line in the list. If an inactive emulator is selected, the emulator starts.

You can create a new emulator by clicking the line + **Create new emulator** at the bottom of the list.

Extension settings

The extension settings are located in the Visual Studio Code storage, which can be called through the menu as follows: **File** → **Preferences** → **Settings**, then **Extensions** → **KasperskyOS**:

`Add Paths`

Add the path `$SDK_PREFIX/toolchain/bin` to the PATH variable. Enabled by default.

`Auto Switch Extension`

Reinstall extension when a new KasperskyOS SDK is selected. Enabled by default.

`Auto Switch Interval`

An interval to check for extension updates in the current KasperskyOS SDK directory (in seconds). Set to `0` to disable. Default is `5`.

`Build Threads Num`

Build threads number. Leave empty to use all available cores.

`Cmake Build Flags`

Additional CMake flags for the build.

`Cmake Config Flags`

Additional configuration flags for running CMake.

`Editor: Auto Open Log Tab`

Automatically open the command log tab. Enabled by default.

`Features:Project`

Enable project features (build/run/test/debug); requires KasperskyOS SDK. Enabled by default.

`Force Clean`

Clean the build directory with the `KOS: Clean build directory` command. Enabled by default.

`Gdbinit File`

Paths to gdbinit files with GDB commands to execute when the debugger is launched. Absolute paths start with `/`. Other paths are relative to workspace root. The ${BUILD_DIR} variable can be used in relative paths only. Default is `['${BUILD_DIR}/.gdbinit.ide']`.

`Nk Lsp Binary`

KasperskyOS NK Language Server binary path.

`Nk Lsp Paths`

Additional search paths for KasperskyOS NK Language Server.

`Sdk Path`

Full path to the installed KasperskyOS SDK.

`Set Build Dir`

Full path to build directory. If unspecified, the `.build/{BuildFeatures}` directory is used.

`Shell Export Variables`

Additional variables to export before running CMake.

`Show Last Logs Num`

Show the number of the last lines of the emulator log. Value `-1` – show all. Default is `500`.

`Show Sdk Env Variables Changed Message`

Show a message when changing SDK that updated environment variables are available in the new terminal. Enabled by default.

`Suppress Extensions Recommendations`

Suppress installation of recommended extensions. Disabled by default.

`Test Arguments`

Arguments for the tests.

`Test Environment Variables`

Environment variables initializations for the tests.

`Test Failures Num`

Maximum number of failed tests to show in Failure details in output window. Leave empty to display all failures.

`Test Failures Only`

Show only failed tests in the output window. Disabled by default.

`Test Files Pattern`

Wildcard pattern to discover test files. Default is `**/{test,tests}/**/*.{c,cpp}`.

`Test No Skipped`

Don't show skipped tests details in the output window. Disabled by default.

`Test Skip Details`

Don't show skipped and failed test details in the output window. Disabled by default.

`Test Threads Num`

Number of threads for multiple test execution. Leave this field blank to use all available cores. (Note 1. The real-time test log is available only when running tests in the same thread. Note 2. VSC will use single thread if only one test is running).

`Test Timeout`

Timeout for each test, in seconds. Set to `0` to use values from test configs. Default is `0`.

`Test Verbose`

Always run tests in verbose mode (print debug information). Disabled by default.

`Trace:Server`

The trace level of the KasperskyOS NK Language Server. Default is `verbose`.

`User SDK Paths`

Additional SDK paths to display in the SDK selection menu.

Building and running examples

Building the examples

The examples are built using the `CMake` build system that is included in KasperskyOS Community Edition SDK.

> Prior to building the examples, make sure that you set the values of the environment variables required for use of the SDK. For more details, refer to Installation and removal.

The code of the examples and build scripts are available at the following path:

`/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples`

> You need to build the examples in a directory in which you have write access. For example, you can do so by copying the directory containing the example into the home directory.

Building the examples to run on QEMU

To build an example, go to the directory containing the example and run the following commands:

```
$ cmake -B build -D CMAKE_TOOLCHAIN_FILE="$KOSCEDIR/toolchain/share/toolchain-*-kos.cmake"
$ cmake --build build --target sim
```

After building the solution image, startup of the example in QEMU is also initiated.

Building the examples to run on Raspberry Pi 4 B or OrangePi RV2

To build an example, go to the directory containing the example and run the following commands:

```
$ cmake -B build -D CMAKE_TOOLCHAIN_FILE="$KOSCEDIR/toolchain/share/toolchain-*-kos.cmake"
$ cmake --build build --target sd-image
```

This creates a file system image for a bootable SD card. The following are loaded into the file system image: the KasperskyOS boot image, the U-Boot bootloader that starts the example, the firmware for the Raspberry Pi 4 B or OrangePiRV2, and all the system programs necessary for running the example. The source code for the U-Boot bootloader and firmware can be downloaded from the website https://github.com. The file system image file `hdd.img` is saved in the directory `<example name>/build`.

Running examples on QEMU

Running examples on QEMU on Linux with a graphical shell

> You need to run the examples in a build directory in which you have write access.

`CMake` is used to run an example on QEMU in Linux with a graphical shell. To run the example, go to the directory containing the example and run the following commands:

```
# Before running the examples, you must set the values of the environment variables.
$ source /opt/KasperskyOS-Community-Edition-<platform>-<version>/common/set_env.sh
# Run the example.
$ cmake -B build -D CMAKE_TOOLCHAIN_FILE="$KOSCEDIR/toolchain/share/toolchain-*-kos.cmake"
$ cmake --build build --target sim
```

Preparing Raspberry Pi 4 B to run examples

Connecting a computer and Raspberry Pi 4 B

To see the output from Raspberry Pi 4 B on a computer and to have debug capabilities, do the following:

1. Connect the pins of the FT232 USB-UART converters to the corresponding GPIO pins of Raspberry Pi 4 B (see the figure below). If debugging is not necessary, all you need to do is connect one USB-UART converter for output.

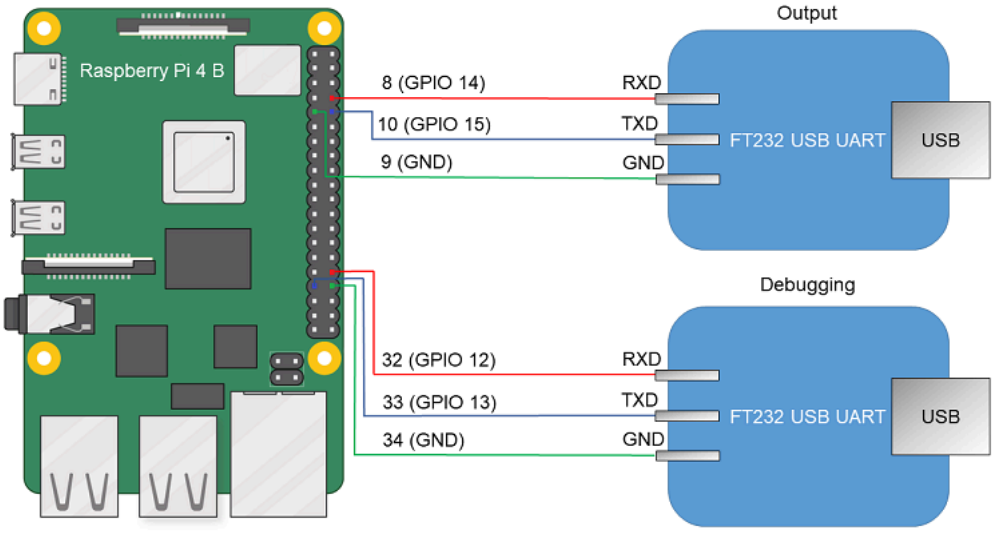


Diagram for connecting USB-UART converters and Raspberry Pi 4 B

2. Connect the computer's USB ports and the USB-UART converters.

3. Install PuTTY or another equivalent program. Configure the settings as follows: `bps = 115200`, `data bits = 8`, `stop bits = 1`, `parity = none`, `flow control = none`. Define the USB port connected to the USB-UART converter used for receiving output from Raspberry Pi 4 B.

To allow a computer and Raspberry Pi 4 B to interact through Ethernet:

1. Connect the network cards of the computer and Raspberry Pi 4 B to a switch or to each other.

2. Configure the computer's network card so that its IP address is in the same subnet as the IP address of the Raspberry Pi 4 B network card (the settings of the Raspberry Pi 4 B network card are defined in the `dhcpcd.conf` file, which is found at the path `<example name>/resources/...`).

Preparing a bootable SD card for Raspberry Pi 4 B

The `hdd.img` image was created when building the example, all you have to do is write the resulting image to the SD card. To do this, connect the SD card to the computer and run the following command:

```
# In the following command, path_to_img is the path to the image file
# [X] is the final character in the name of the SD card block device.
$ sudo dd of=/dev/sd[X] if=path_to_img status=progress bs=64K conv=fsync
```

Preparing OrangePi RV2 to run examples

Connecting a computer and OrangePi RV2

To see the output from OrangePi RV2 on a computer, do the following:

1. Connect the pins of the FT232 USB-UART converter to the corresponding GPIO pins of OrangePi RV2 (see the figure below).

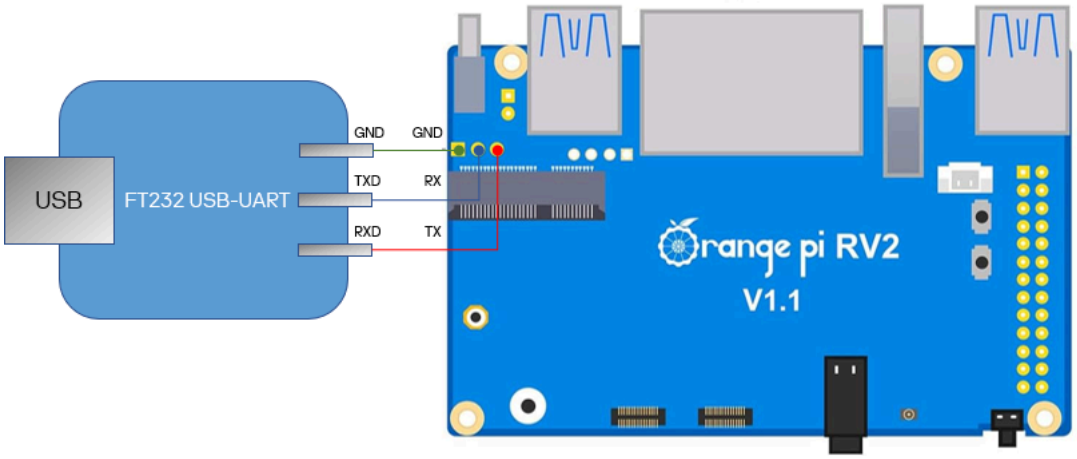


Diagram for connecting the FT232 USB-UART converter and OrangePi RV2

2. Connect the computer's USB ports and the FT232 USB-UART converter.

3. Install PuTTY or another equivalent program. For the main output, configure the settings as follows:

   `bps = 115200, data bits = 8, stop bits = 1, parity = none, flow control = none`

   Define the USB port that is connected to the FT232 USB-UART converter used for receiving output from OrangePi RV2.

To allow a computer and OrangePi RV2 to interact over Ethernet, do the following:

1. Connect the network cards of the computer and OrangePi RV2 to a switch or to each other.

2. Configure the computer's network card so that its IP address is in the same subnet as the IP address of the OrangePi RV2 network card (the settings of the OrangePi RV2 network card are defined in the `dhcpcd.conf` file, which is located at the path `<example name>/resources/...`).

Preparing a bootable SD card for OrangePi RV2

The `hdd.img` image was created when building the example, all you have to do is write the resulting image to the SD card. To do this, connect the SD card to the computer and run the following command:

```
# In the following command, path_to_img is the path to the image file
# [X] is the final character in the name of the SD card block device.
$ sudo dd of=/dev/sd[X] if=path_to_img status=progress bs=64K conv=fsync
```

Running examples on Raspberry Pi 4 B or OrangePi RV2

To run an example on Raspberry Pi 4 B or OrangePi RV2, do the following:

1. Go to the directory with the example and build the example.
2. Make sure that Raspberry Pi 4 B or OrangePi RV2 and the bootable SD card are prepared to run examples.
3. Connect the bootable SD card to the Raspberry Pi 4 B or OrangePi RV2.
4. Supply power to the Raspberry Pi 4 B or OrangePi RV2 and wait for the example to start.

   The output displayed on the computer connected to the Raspberry Pi 4 B or OrangePi RV2 indicates that the example has started.

Development for KasperskyOS

Starting processes

ServiceManager system program

The ServiceManager system program is provided in the SDK and is responsible for installing applications from KPA packages at the first startup of a solution and running those applications at each subsequent startup of the solution.

For more details on starting a solution, see Architecture and startup of a KasperskyOS-based solution.

The ServiceManager system program uses the following configuration files in JSON format:

- The initial_install.json file must contain a list of KPA packages that install when KasperskyOS is started for the first time.
- The system_services.json file contains a list of installed programs that start each time KasperskyOS starts.

The initial_install.json and system_services.json files must be created in the project directory and, when building the solution, they must be added to the root file system directory defined by the CMake variable `${ROOTFS_DIR}${MINIMAL_KOS_INITIAL_SETTINGS_DIR}`.

When building the solution, KPA packages of applications that will be installed by the ServiceManager system program must be added to the root file system directory defined by the CMake variable `${ROOTFS_DIR}${MINIMAL_KOS_INITIAL_PACKAGES_DIR}`.

For details, see the ./einit/CMakeLists.txt build script files in the SDK examples.

Syntax of the initial_install.json file

This file contains a `packages` object with an array of entries corresponding to each package being installed. Each entry contains a `name` string attribute, which contains the name (without the KPA extension) of the KPA package file to be installed.

Example:

initial_install.json

```
{
    "packages": [
        {
            "name": "Hello"
        }
    ]
}
```

Syntax of the system_services.json file

This file contains the `services` object with an array of entries corresponding to each program being run. Each entry contains the following:

- `packageId` string attribute containing the name of the KPA package to be installed. The name of the KPA package is specified in the `ID` parameter when building the package using the CMake command add_kpa_package().
- `runConfigurationId` string attribute containing the name of the program startup configuration. The name of the program startup configuration is specified in the `ID` parameter when adding the program startup configuration using the CMake command add_kpa_component().

Example:

system_services.json

```
{
    "services": {
        "Hello": {
          "packageId": "Hello",
          "runConfigurationId": "default"
        }
    }
}
```

Starting processes using the system program ExecutionManager

The ExecutionManager component provides a C++ interface for creating, starting and stopping processes in solutions that are based on KasperskyOS.

> The interface of the ExecutionManager component is not suitable for use in code that is written in C. To manage processes in C, use the task.h API of the libkos library.

You can use the ExecutionManager component to start processes from the following:

- Executable file with a known location in the file system.
- Executable file installed by the PackageManager component in a KasperskyOS-based solution from a KPA package.

The ExecutionManager component API is built on top of IPC and helps simplify program development. ExecutionManager is a separate system program that is accessed through IPC. However, developers are provided with a client library that eliminates the necessity of directly using IPC calls.

The programming interface of the ExecutionManager component is described in the article titled "ExecutionManager component".

ExecutionManager component usage scenario

Hereinafter "the client" refers to the application that uses the ExecutionManager component API to manage other applications.

The ExecutionManager system program is supplied as part of the KasperskyOS boot image and starts automatically when the solution is started.

The typical usage scenario for the ExecutionManager component includes the following steps:

1. Link the client executable file to the client proxy library of ExecutionManager by adding the following command to the `CMakeLists.txt` file for building the client:

   ```
   target_link_libraries (<name of the CMake target for building the client>
   execution_manager::execution_manager_proxy)
   ```

2. Add permissions for the necessary events to the solution security policy description:

   a. To enable the ExecutionManager program to run other processes, the solution security policy must allow the following interactions for the `execution_manager.ExecMgrEntity` process class:

   - Security events of the `execute` type for all classes of processes that will be run.
   - Access to all endpoints of the VFS program.
   - Access to all endpoints of the BlobContainer program (their descriptions are located in the directory `sysroot-*-kos/include/kl/bc` from the SDK).
   - Access to the core endpoints `Sync`, `Task`, `VMM`, `Thread`, `HAL`, `Handle`, `FS`, `Notice`, `CM` and `Profiler` (their descriptions are located in the directory `sysroot-*-kos/include/kl/core` from the SDK).

   b. To enable a client to call the ExecutionManager program, the solution security policy must allow the following interactions for the client process class:

   - Access to the appropriate endpoints of the ExecutionManager program (their descriptions are located in the directory `sysroot-*-kos/include/kl/execution_manager` from the SDK).

3. Use of the ExecutionManager program API in the client code.

   Use the header file `component/execution_manager/kos_ipc/execution_manager_proxy.h` for this. For more details, refer to "ExecutionManager component".

kl.ProductRomfsLoader initializing program and product_init.yaml file

kl.ProductRomfsLoader initializing program

> You are advised to use KPA packages and the ServiceManager system program to run applications.

If a KasperskyOS-based solution uses programs that for some reason cannot be packed into KPA packages, you can start these programs by using the kl.ProductRomfsLoader system program, which starts programs based on the init description. You are advised to use this approach only if it is absolutely necessary. For more details, see the examples included in the SDK.

The kl.ProductRomfsLoader program starts programs in accordance with the init description (the product_init.yaml file). This program startup method is not recommended but is used to start some system programs that do not support startup from KPA packages.

Init description (product_init.yaml)

An init description contains data in YAML format. This data identifies the following:

- Processes that will be started by the kl.ProductRomfsLoader system program

  The order in which processes are started does not depend on the order in which they are listed in the init description, but the order can be defined by using the `before` and `after` keys (see the table below).

- IPC channels that are created when the solution starts and are used by processes to interact with each other.

This data consists of a dictionary with the `entities` key containing a list of dictionaries of processes (see Example init descriptions). Process dictionary keys are presented in the table below.

You can create an init description manually or automatically. In the second case, you need to prepare the `product_init.yaml.in` template from which the build system will create an init description.

Process dictionary keys in an init description

Key	Required	Value
`name`	Yes	Process class name (from the EDL description).
`task`	No	Process name. If this name is not specified, the process class name will be used. Each process must have a unique name. You can start multiple processes of the same class if they have different names.
`path`	No	Name of the executable file in ROMFS (in the solution image). If this name is not specified, the process class name (without prefixes and dots) will be used. For example, processes of the `Client` and `net.Client` classes for which an executable file name is not specified will be started from the `Client` file. You can start multiple processes from the same executable file.

Key	Required	Value
`connections`	No	Process IPC channel dictionaries list. This list defines the statically created IPC channels whose client IPC handles will be owned by the process. The list is empty by default. (In addition to statically created IPC channels, processes can also use dynamically created IPC channels.)
`args`	No	List of program startup parameters (`main()` function parameters). The maximum size of one item on the list is 1024 bytes.
`env`	No	Dictionary of program environment variables. The keys in this dictionary are the names of environment variables. The maximum size of an environment variable value is 65524 bytes.
`before`	No	Determines the order in which processes are started. You must specify a list of names of processes that should be started after starting this process.
`after`	No	Determines the order in which processes are started. You must specify a list of names of processes that must be started before starting this process.

`wait_for`	No	Sign used to determine that the process has started successfully.
		Syntax options:
		`wait_for: <event_name>`
		or
		`wait_for:`
		`event: <event_name>`
		`timeout: <timeout_value><timeout_suffix>`
		or
		`wait_for: { event: <event_name>, timeout: <timeout_value><timeout_suffix> }`
		Parameter `<event_name>` can take the following values:
		• `start` – wait for the `main()` function to start.
		• `ready` – wait for the `em_notify_ready()` signal from the running process. The function is declared in the header file `sysroot-*-kos/include/em_transport/em_transport.h`.
		• `exit_success` – wait for the successful completion of the process.
		The `<timeout_value>` parameter is an integer representing the time to wait for a sign of successful startup of the process. Default value: 3 minutes.
		The `<timeout_suffix>` parameter is specified without a space after the `<timeout_value>` parameter and can take the following values:
		• `ms`, `msec` – milliseconds
		• `s`, `sec` – seconds
		• `m`, `min` – minutes
		Usage examples:
		`wait_for: ready`
		`wait_for: { event: ready, timeout: 100ms }`
		`wait_for:`
		`event: ready`
		`timeout: 100ms`
		For more details, refer to Example init descriptions.

Process IPC channel dictionary keys are presented in the table below.

IPC channel dictionary keys in an init description

Key	Required	Value
`id`	Yes	IPC channel name, which can be set as a fixed value or as a link such as `{var: <constant name>, include: <path to header file>}`.
`target`	Yes	Name of the process that will own the server handle of the IPC channel.

The init description may also contain the `mount_romfs` key, which specifies the name of the mount point of the additional `romfs.1` file system before starting the processes. This allows you to start processes from binary files located in the `romfs.1` file system. The `romfs.1` file system will be unmounted after the `Einit` process ends. Usage example:

```
mount_romfs:
  "romfs.1": "romfs/1"
```

Example init descriptions

This section provides examples of init descriptions that demonstrate various aspects of starting processes.

Starting a client and server and creating an IPC channel between them

In this example, a process of the `Client` class and a process of the `Server` class are started. The names of the processes are not specified, so they will match the names of their respective process classes. The names of the executable files are not specified, so they will also match the names of their respective process classes. The processes will be connected by an IPC channel named `server_connection`.

product_init.yaml (snippet)

```
entities:
- name: Client
  connections:
  - target: Server
    id: server_connection
- name: Server
```

Starting processes from defined executable files

This example will start a `Client`-class process from the executable file named `cl`, a `ClientServer`-class process from the executable file named `csr`, and a `MainServer`-class process from the executable file named `msr`. The names of the processes are not specified, so they will match the names of their respective process classes.

product_init.yaml (snippet)

```
entities:
- name: Client
  path: cl
- name: ClientServer
  path: csr
- name: MainServer
  path: msr
```

Starting two processes from the same executable file

This example will start a `Client`-class process from the executable file named `Client`, and two processes of the `MainServer` and `BkServer` classes from the executable file named `srv`. The names of the processes are not specified, so they will match the names of their respective process classes.

product_init.yaml (snippet)

```
entities:
- name: Client
- name: MainServer
```

```
    path: srv
  - name: BkServer
    path: srv
```

Starting two servers of the same class and a client, and creating IPC channels between the client and servers

This example will start a `Client`-class process (named `Client`) and two `Server`-class processes named `UserServer` and `PrivilegedServer`. The client will be connected to the servers via IPC channels named `server_connection_us` and `server_connection_ps`. The names of the executable files are not specified, so they will match the names of their respective process classes.

product_init.yaml (snippet)

```
entities:
- name: Client
  connections:
  - id: server_connection_us
    target: UserServer
  - id: server_connection_ps
    target: PrivilegedServer
- task: UserServer
  name: Server
- task: PrivilegedServer
  name: Server
```

Setting the startup parameters and environment variables of programs

This example will start a `VfsFirst`-class process (named `VfsFirst`) and a `VfsSecond`-class process (named `VfsSecond`). The program that will run in the context of the `VfsFirst` process will be started with the parameter `-f /etc/fstab`, and will receive the `ROOTFS` environment variable with the value `ramdisk0,0 / ext2 0` and the `UNMAP_ROMFS` environment variable with the value `1`. The program that will run in the context of the `VfsSecond` process will be started with the `-l devfs /dev devfs 0` parameter. The names of the executable files are not specified, so they will match the names of their respective process classes.

product_init.yaml (snippet)

```
entities:
- name: VfsFirst
  args:
  - -f
  - /etc/fstab
  env:
    ROOTFS: ramdisk0,0 / ext2 0
    UNMAP_ROMFS: 1
- name: VfsSecond
  args:
  - -l
  - devfs /dev devfs 0
```

Kernel startup parameters that are passed to programs as startup parameters and environment variables have the following format:

```
<name>=<value>
```

To pass parameters via the kernel command line to QEMU, use the QEMU startup parameter `-append`.

Example of setting kernel startup parameters on QEMU using the `CMake` command `generate_qemu_target()`:

```
set(QEMU_FLAGS "-append \"ROOTFS=/opt/data test_switch=-b FSTAB=/etc/fstab MOTD=Hello! timeout=30 FS=/home\"")
generate_qemu_target(qemu
                     ...
                     QEMU_FLAGS "${QEMU_FLAGS}")
```

To find out how to pass parameters via the command line of the kernel on a hardware platform, please refer to the documentation for the loader that you are using.

Setting the startup order of processes and the signs of successful startup of a process

In this example:

- The ProductRomfsLoader process will mount the additional `romfs.1` file system to the `romfs/1` mount point.
- The `Server` class process will be started from the `server` file in the main ROMFS file system before processes of the `Client_A` and `Client_B` classes. The `Einit` program will wait for the `em_notify_ready()` signal from the `Server`.
- The `Client_A` class process will be started from the `romfs/1/client_a` file after the `Server` class process. The ProductRomfsLoader process will wait for the `main()` function to run successfully in `Client_A`.
- The `Client_B` class process will be started from the `romfs/1/client_b` file after the `Server` and `Client_A` class process. The ProductRomfsLoader process will wait five minutes for `Client_B` to finish successfully.

product_init.yaml (snippet)

```
mount_romfs:
  "romfs.1": "romfs/1"
entities:
- name: Server
  path: "server"
  wait_for: ready
  before:
   - Client_A
   - Client_B
- name: Client_A
  path: "romfs/1/client_a"
  wait_for: start
  after:
   - Server
- name: Client_B
  path: "romfs/1/client_b"
  wait_for: { event: exit_success, timeout: 5min }
  after:
   - Client_A
   - Server
```

File systems and network

In KasperskyOS, operations with file systems and the network are executed via a separate system program that implements a virtual file system (VFS).

In the SDK, the VFS component consists of a set of executable files, libraries, formal specification files, and header files. For more details, see the Contents of the VFS component section.

The main scenario of interaction with the VFS system program includes the following:

1. An application connects via IPC channel to the VFS system program and then links to the client library of the VFS component during the build.

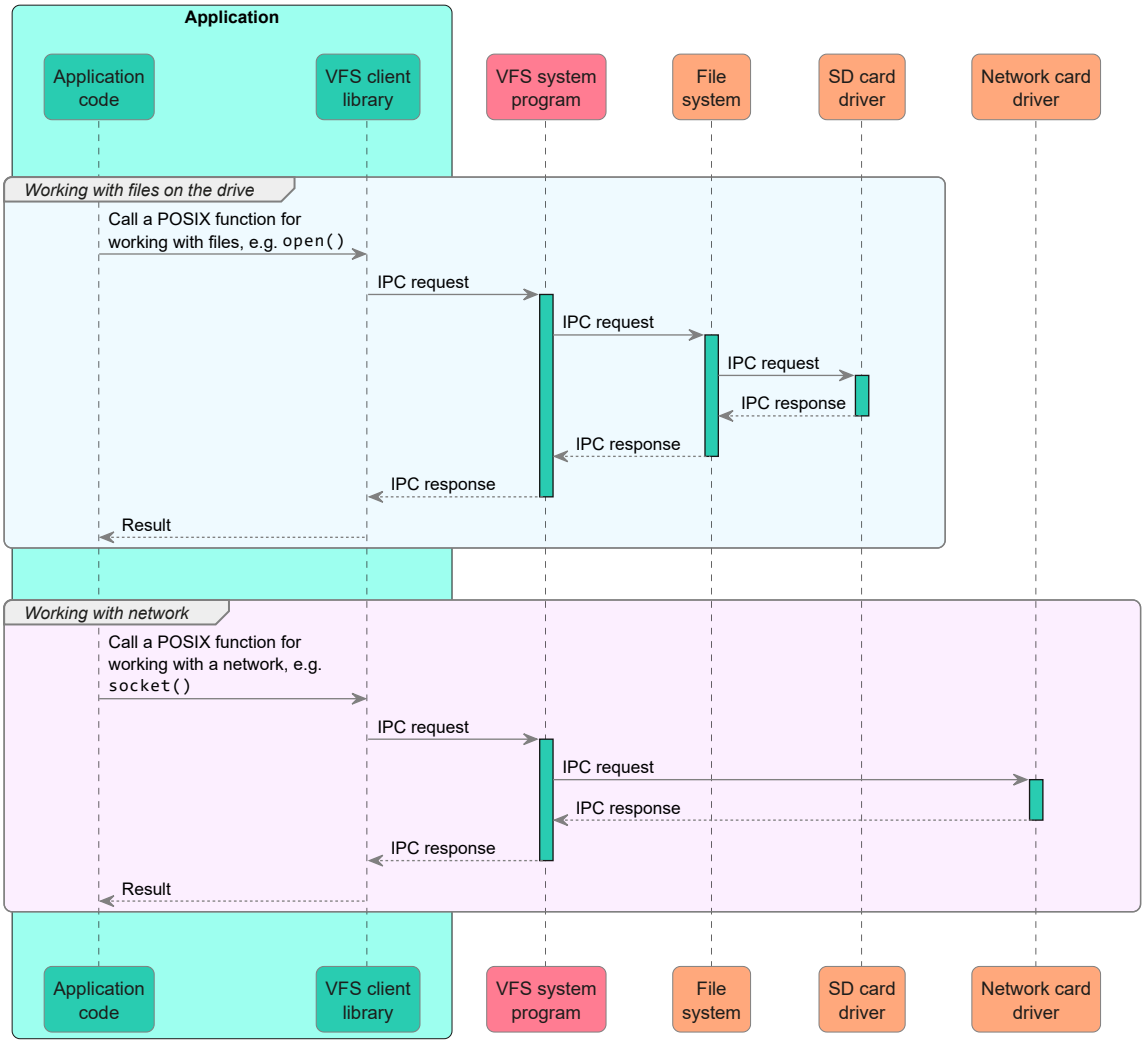
2. In the application code, POSIX calls for working with file systems and the network are converted into client library function calls.

   > Input and output to file handles for standard I/O streams (stdin, stdout and stderr) are also converted into queries to the VFS. If the application *is not linked* to the client library of the VFS component, printing to stdout is not possible. If this is the case, you can only print to the standard error stream (stderr), which in this case is performed via special methods of the KasperskyOS kernel without using VFS.

3. The client library makes IPC requests to the VFS system program.

4. The VFS system program receives an IPC requests and calls the corresponding file system implementations (which, in turn, may make IPC requests to device drivers) or network drivers.

5. After the request is handled, the VFS system program responds to the IPC requests of the application.



Main scenario of interaction with the VFS system program

Building a VFS executable file

When building a solution, you can build a modified VFS program by adding the required functionality to it. For more details, refer to Building a VFS executable file.

Using multiple VFS programs

Multiple copies of the VFS system program can be added to a solution for the purpose of separating the data streams of different system programs and applications. You can also separate the data streams within one application. For more details, refer to Using VFS backends to separate data streams.

Adding VFS functionality to an application

The complete functionality of the VFS component can be included in an application, thereby avoiding the need to pass each request via IPC. For more details, refer to Including VFS functionality in a program.

However, use of VFS functionality via IPC enables the solution developer to do the following:

- Use a solution security policy to control method calls for working with the network and file systems.
- Connect multiple client programs to one VFS program.
- Connect one client program to two VFS programs to separately work with the network and file systems.

Working with the VFS system program

This section provides information about working with the VFS system program.

Contents of the VFS component

The VFS component implements the virtual file system. In the KasperskyOS SDK, the VFS component consists of a set of executable files, libraries, formal specification files and header files that enable the use of file systems and/or a network stack.

VFS libraries

The `vfs` CMake package contains the following libraries:

- `vfs::lib_fs` contains implementations of the devfs, ramfs and ROMFS file systems, and adds implementations of other file systems to VFS.
- `vfs::lib_fs` contains the implementation of the devfs file system and the network stack.
- `vfs::client` is the client transport library that converts local calls into IPC requests to VFS and receives IPC responses.
- `vfs::server` is the VFS server transport library that receives IPC requests, converts them into local calls, and sends IPC responses.
- `vfs::local` is used to include VFS functionality in a program.
- `vfs::relay` is used to implement the VFS relay mechanism.

VFS executable files

The `precompiled_vfs` CMake package contains the following executable files:

- `VfsRamFs`
- `VfsSdCardFs`
- `VfsNet`

The `VfsRamFs` and `VfsSdCardFs` executable files include the `vfs::server`, `vfs::lib_fs`, `fs::vfat`, and `fs::lwext4` libraries. The `VfsNet` executable file includes the `vfs::server` and `vfs::lib_net` libraries.

Each of these executable files has its own default values for startup parameters and environment variables.

If necessary, you can independently build a VFS executable file with the necessary functionality.

Formal specification files and header files of VFS

The `sysroot-*-kos/include/kl` directory from the KasperskyOS SDK contains the following VFS files:

- Formal specification files `VfsRamFs.edl`, `VfsSdCardFs.edl`, `VfsNet.edl` and `VfsEntity.edl`, and the header files generated from them.
- Formal specification file `Vfs.cdl` and the header file `Vfs.cdl.h` generated from it.
- Formal specification files `Vfs*.idl` and the header files generated from them.

Libc library API supported by VFS

VFS functionality is available to programs through the API provided by the `libc` library.

The functions implemented by the `vfs::lib_fs` and `vfs::lib_net` libraries are shown in the tables below. The `*` character denotes the functions that are optionally included in the `vfs::lib_fs` library (depending on the library build parameters).

Functions implemented by the `vfs::lib_fs` library

mount()	unlink()	ftruncate()	lsetxattr()*
umount()	rmdir()	chdir()	fsetxattr()*
open()	mkdir()	fchdir()	getxattr()*
openat()	mkdirat()	chmod()	lgetxattr()*
read()	fcntl()	fchmod()	fgetxattr()*
readv()	statvfs()	fchmodat()	listxattr()*
write()	fstatvfs()	chroot()	llistxattr()*
writev()	getvfsstat()	fsync()	flistxattr()*
stat()	pipe()	fdatasync()	removexattr()*
lstat()	futimens()	pread()	lremovexattr()*
fstat()	utimensat()	pwrite()	fremovexattr()*
fstatat()	link()	sendfile()	acl_set_file()*
lseek()	linkat()	getdents()	acl_get_file()*
close()	symlink()	sync()	acl_delete_def_file()*
rename()	symlinkat()	ioctl()	
renameat()	unlinkat()	setxattr()*	

Functions implemented by the `vfs::lib_net` library

read()	bind()	getsockname()	recvfrom()
readv()	listen()	gethostbyname()	recvmsg()
write()	connect()	getnetbyaddr()	send()
writev()	accept()	getnetbyname()	sendto()
fstat()	poll()	getnetent()	sendmsg()
close()	shutdown()	setnetent()	ioctl()
fcntl()	getnameinfo()	endnetent()	sysctl()
fstatvfs()	getaddrinfo()	getprotobyname()	
pipe()	freeaddrinfo()	getprotobynumber()	
futimens()	getifaddrs()	getsockopt()	
socket()	freeifaddrs()	setsockopt()	
socketpair()	getpeername()	recv()	

> If there is no implementation of a called function in VFS, the `EIO` error code is returned.

Creating a static IPC channel to VFS

In this example, the `Client` process uses the file systems and network stack, and the `VfsFsnet` process handles the IPC requests of the `Client` process related to the use of file systems and the network stack. This approach is utilized when there is no need to separate data streams related to file systems and the network stack.

The IPC channel name must be assigned by the `_VFS_CONNECTION_ID` macro defined in the header file `sysroot-*-kos/include/vfs/defs.h` from the KasperskyOS SDK.

Init description

product_init.yaml

```
- name: Client
  connections:
  - target: VfsFsnet
    id: {var: _VFS_CONNECTION_ID, include: vfs/defs.h}

- name: VfsFsnet
```

Building a VFS executable file

When building a VFS executable file, you can include whatever specific functionality is required in this file, such as:

- File systems
- Network stack
- Network driver

For example, you will need to build a "file version" and a "network version" of VFS to separate data streams related to file systems and the network stack. In some cases, you need to include a network stack and file systems in the VFS.

Building a "file version" of the VFS program

In this example, the VFS program contains only an lwext4 file system implementation (see the figure below). The `vfs.c` implementation file is compiled, and the `vfs_server`, `vfs_fs` and `lwext4` libraries are linked: For more details, see vfs_extfs example.

CMakeLists.txt

```
project (vfsfs)

# Set compile flags
project_header_default ("STANDARD_GNU_17:YES" "STRICT_WARNINGS:NO")

add_executable (VfsFs "src/vfs.c")

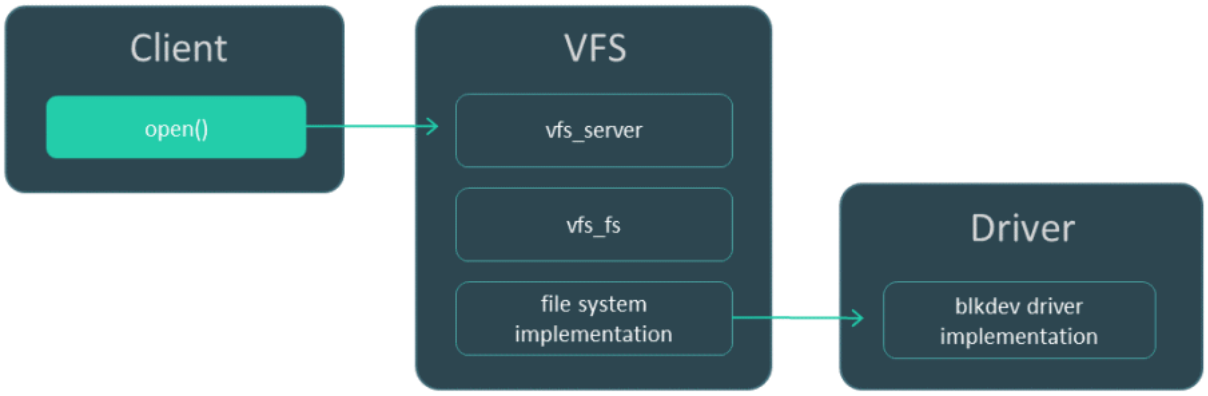
# Include packages required for building the solution
find_package (vfs REQUIRED)
find_package (LWEXT4 REQUIRED)
find_package (blkdev REQUIRED)
```

```
find_package (ramdisk REQUIRED)

# Linking with VFS libraries
target_link_libraries (VfsFs
                       ${vfs_SERVER_LIB}
                       ${LWEXT4_LIB}
                       ${vfs_FS_LIB})

# Prepare VFS to connect to the ramdisk driver process
set_target_properties (VfsFs PROPERTIES ${blkdev_ENTITY}_REPLACEMENT ${ramdisk_ENTITY})
```

> A block device driver cannot be linked to VFS and therefore must be run in the context of a separate process.



Interaction between three processes: client, "file version" of VFS, and block device driver.

Building a "network version" of the VFS program containing a network driver

In this example, the VFS program contains implementations of the network stack and network driver (see the figure below). The `vfs.c` implementation file is compiled and the `vfs_server`, `vfs_implementation` and `dnet_implementation` libraries are linked: For more details, see net_with_separate_vfs example.

CMakeLists.txt

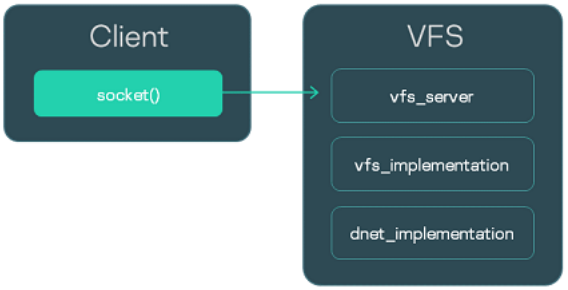
```
project (vfsnet)

# Set compile flags
project_header_default ("STANDARD_GNU_17:YES" "STRICT_WARNINGS:NO")

add_executable (VfsNet "src/vfs.c")
# Include packages required for building the solution
find_package (vfs REQUIRED)
find_package (dnet REQUIRED)
find_package (blkdev REQUIRED)

# Linking with VFS libraries
target_link_libraries (VfsNet
                       ${vfs_SERVER_LIB}
                       ${vfs_IMPLEMENTATION_LIB}
                       ${dnet_IMPLEMENTATION_LIB})
```

```
# Disconnect the block device driver
set_target_properties (VfsNet PROPERTIES ${blkdev_ENTITY}_REPLACEMENT "")
```



Interaction between the Client process and the process of the "network version" of VFS

Building the "network version" of the VFS program without a network driver

In this example, the "network version" of the VFS program does not contain a network driver implementation (see the figure below). (The network driver will run in the context of a separate process. Interaction with the driver will occur via IPC using the `dnet_client` library.) The `vfs.c` implementation file is compiled and the `vfs_server`, `vfs_implementation` and `dnet_client` libraries are linked: For more details, see net_with_separate_vfs example.

CMakeLists.txt

```
project (vfsnet)

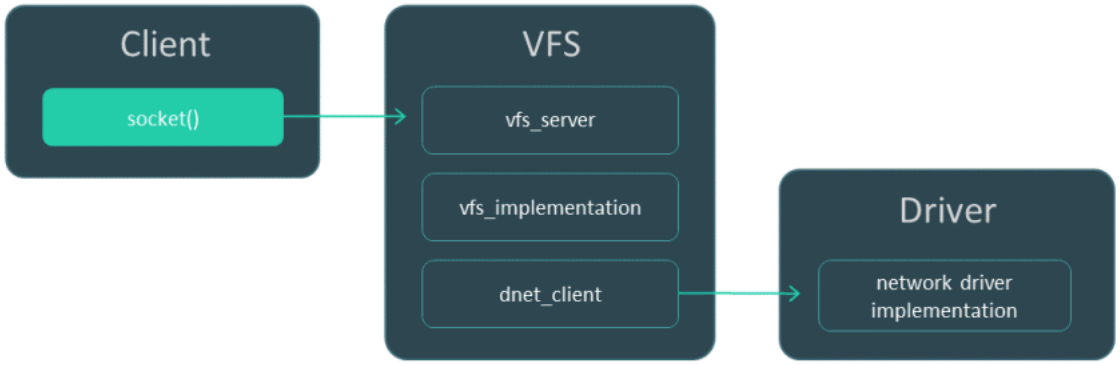
# Set compile flags
project_header_default ("STANDARD_GNU_17:YES" "STRICT_WARNINGS:NO")

add_executable (VfsNet "src/vfs.c")

# Include packages required for building the solution
find_package (vfs REQUIRED)
find_package (dnet REQUIRED)
find_package (blkdev REQUIRED)

# Linking with VFS libraries
target_link_libraries (VfsNet
                       ${vfs_SERVER_LIB}
                       ${vfs_IMPLEMENTATION_LIB}
                       ${dnet_CLIENT_LIB})

# Disconnect the block device driver
set_target_properties (VfsNet PROPERTIES ${blkdev_ENTITY}_REPLACEMENT "")
```



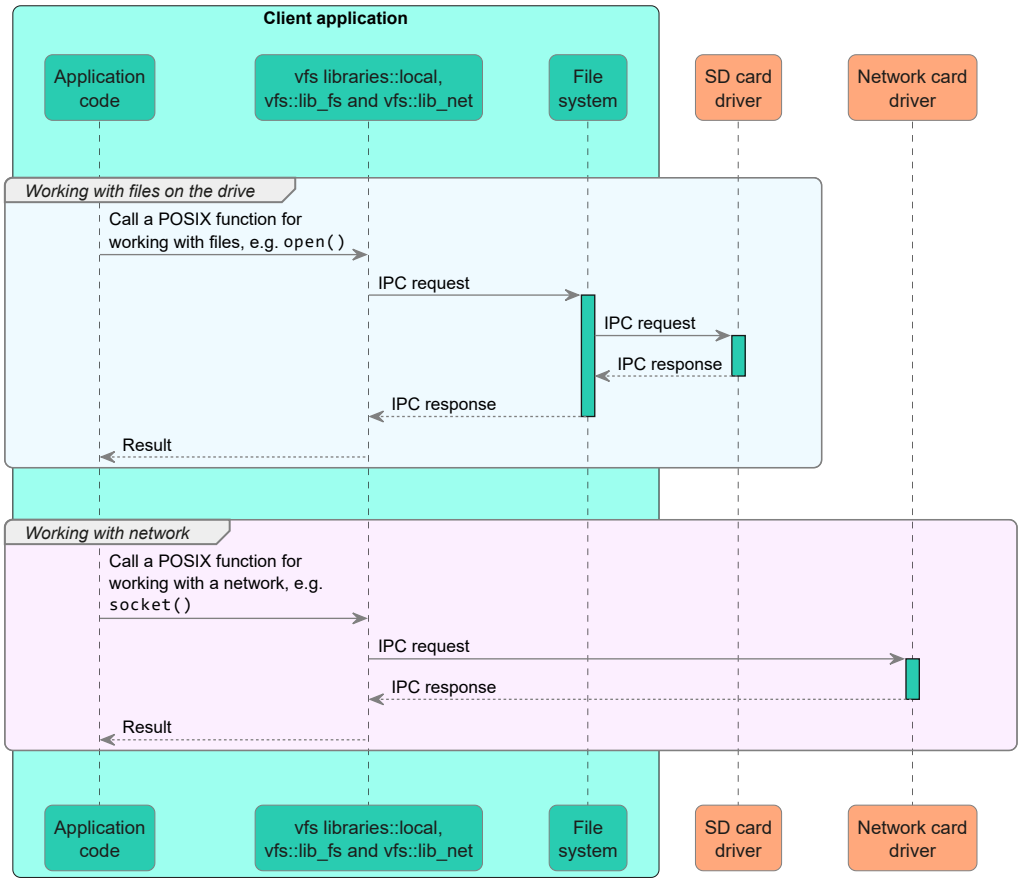
Interaction between three processes: client, "network version" of VFS, and network driver.

Building a "full version" of the VFS program

If you need to include implementations of a network stack and file systems in the VFS program, you must compile the `vfs.c` implementation file and link the `vfs_server`, `vfs_implementation`, and `dnet_implementation` libraries (or the `dnet_client` library if the network driver will run in the context of a separate process), and link the libraries for implementing file systems.

Including VFS functionality in a program

In this example, the `Client` program includes the VFS program functionality for working with the files and network stack (see the figure below). For more details, refer to embedded_vfs example.



VFS component libraries in a program

The `client.c` implementation file is compiled and linked to the `vfs::local`, `vfs::lib_fs`, `vfs::lib_net`, and `dnet_drivers::drivers` libraries:

CMakeLists.txt

```
project (client)

include (platform/nk)

find_package (vfs REQUIRED)
find_package (dnet REQUIRED)
find_package (blkdev REQUIRED)
find_package (ramdisk REQUIRED)
find_package (LWEXT4 REQUIRED)

# Set compile flags
project_header_default ("STANDARD_GNU_17:YES" "STRICT_WARNINGS:NO")
```

```
# Generates the Client.edl.h file
nk_build_edl_files (client_edl_files NK_MODULE "client" EDL "${CMAKE_SOURCE_DIR}/resources/edl/Client.edl")

add_executable (Client "src/client.c")
add_dependencies (Client client_edl_files)

# Linking with VFS libraries
target_link_libraries (Client vfs::local ${vfs_IMPLEMENTATION_LIB} dnet_class::client ${LWEXT4_LIB}
blkdev::client)

set_target_properties (Client PROPERTIES ${blkdev_ENTITY}_REPLACEMENT ${ramdisk_ENTITY})
```

VFS startup parameters and environment variables

VFS program startup parameters

- `-l <entry in fstab format>`

  The startup parameter `-l` mounts the defined file system.

- `-f <path to fstab file>`

  The parameter `-f` mounts the file systems specified in the `fstab` file. If the `UNMAP_ROMFS` environment variable is not defined, the `fstab` file will be sought in the ROMFS image. If the `UNMAP_ROMFS` environment variable is defined, the `fstab` file will be sought in the file system defined through the `ROOTFS` environment variable.

Examples of using VFS program startup parameters

Environment variables of the VFS program

- `UNMAP_ROMFS`

  If the `UNMAP_ROMFS` environment variable is defined, the ROMFS image will be deleted from memory. This helps conserve memory. When using the startup parameter `-f`, it also provides the capability to search for the `fstab` file in the file system defined through the `ROOTFS` environment variable instead of searching the ROMFS image.

  Example of using the UNMAP_ROMFS environment variable

- `ROOTFS = <entry in fstab format>`

  The `ROOTFS` environment variable mounts the defined file system to the root directory. When using the startup parameter `-f`, a combination of the `ROOTFS` and `UNMAP_ROMFS` environment variables provides the capability to search for the `fstab` file in the file system defined through the `ROOTFS` environment variable instead of searching the ROMFS image.

  Example of using the ROOTFS environment variable

- `VFS_SERVER_MAX_THREADS_PER_CLIENT`

  The `VFS_SERVER_MAX_THREADS_PER_CLIENT` variable defines the number of VFS process threads allocated to one IPC channel.

- `VFS_NETWORK_BACKEND=<VFS backend name>:<name of the IPC channel to the VFS process>`

The `VFS_NETWORK_BACKEND` environment variable defines the VFS backend for working with the network stack. Additionally:

- You can specify the name of the standard VFS backend: `client` (for a program that runs in the context of a client process), `server` (for a VFS program that runs in the context of a server process), `local`, `relay` (when using the VFS relay mechanism), and the name of a custom VFS backend.
- You can specify more than one IPC channel by separating them with a comma.
- If the `local` VFS backend is used, the name of the IPC channel is not specified (`VFS_NETWORK_BACKEND=local:`).
- When using dynamic IPC channels, instead of the name of the IPC channel to the VFS process, you must specify the name of the VFS process, for example: `VFS_NETWORK_BACKEND: client:kl.VfsNet`.
- When using the standard `client` backend, you can specify different values for the `VFS_NETWORK_BACKEND` and `VFS_FILESYSTEM_BACKEND` variables. However, when using the standard `server`, `relay`, or `local` backends, you must either specify the value of only one of these variables or specify the same values for both variables.
- If the value of the `VFS_NETWORK_BACKEND` variable is not specified but the value of the `VFS_FILESYSTEM_BACKEND` variable is specified, the use of network operations will lead to termination of the VFS program.

- `VFS_FILESYSTEM_BACKEND=<VFS backend name>:<name of the IPC channel to the VFS process>`

  The `VFS_FILESYSTEM_BACKEND` environment variable defines the VFS backend for working with file systems. Additionally:

  - You can specify the name of the standard VFS backend: `client` (for a program that runs in the context of a client process), `server` (for a VFS program that runs in the context of a server process), `local`, `relay` (when using the VFS relay mechanism), and the name of a custom VFS backend.
  - You can specify more than one IPC channel by separating them with a comma.
  - If the `local` VFS backend is used, the name of the IPC channel is not specified (`VFS_NETWORK_BACKEND=local:`).
  - When using dynamic IPC channels, instead of the name of the IPC channel to the VFS process, you must specify the name of the VFS process, for example: `VFS_FILESYSTEM_BACKEND: client: kl.VfsFs`.
  - When using the standard `client` backend, you can specify different values for the `VFS_NETWORK_BACKEND` and `VFS_FILESYSTEM_BACKEND` variables. However, when using the standard `server`, `relay`, or `local` backends, you must either specify the value of only one of these variables or specify the same values for both variables.
  - If the value of the `VFS_FILESYSTEM_BACKEND` variable is not specified, but the value of the `VFS_NETWORK_BACKEND` variable is specified, the use of file operations will lead to termination of the VFS program.

- `VFS_GENERAL_PIPE_BUFFER_SIZE=<byte count>`

  The environment variable `VFS_GENERAL_PIPE_BUFFER_SIZE` defines the maximum size of the buffer (in bytes) that can be used by an unnamed channel (pipe) to store transmitted data. This size can be subsequently changed by using the `F_SETPIPE_SZ` macro of the `fcntl()` function. The minimum value of the environment variable is equal to one memory page (`PAGE_SIZE`), maximum value: `1048576` bytes, default value: `65536` bytes.

- `RUMP_NETEN_NAME_POLICY=<list of network interface naming policies>`

  The `RUMP_NETEN_NAME_POLICY` environment variable defines the list of policies for naming network interfaces for VFS. Policies are listed in their order of use, separated by commas (for example, `RUMP_NETEN_NAME_POLICY="path,alias,default"`). Possible names of policies and their descriptions are presented in the table below. If the first policy specified is not applicable to the current device (see the Policy applicability column in the table below), the second policy is used, and so on. The most recent policy is applied. If no policies are specified, the `default` policy is applied.

  Network interface naming policies

Policy	Name examples	Naming template	Policy applicability
`default`	`en0` – the first Ethernet interface. `en1` – the second Ethernet interface. `wl0` – the first Wi-Fi interface. `wwan0` – the first modem interface.	`<prefix><index>`: • `<prefix>` – defines the type of interface. • `<index>` – sequence number of the device with the `<prefix>` interface type, starting from `0`.	For all devices.
`path`	`enp1s4f0` – a PCIE device with PCI segment `1`, PCI bus `4`, and PCI function `0`.	`<prefix>p<segment>s<bus>f<function>`: • `<prefix>` – defines the type of interface. • `<segment>` – PCI segment number. • `<bus>` – PCI bus number. • `<function>` – PCI function number.	Only for PCIE devices
`alias`	`ethernet0` – the first Ethernet interface. `ethernet1` – the second Ethernet interface. `wlan0` – the first Wi-Fi interface. `modem0` – the first modem interface.	The device alias defined in the DTB (Device Tree Blob) is used.	Only if a device alias is defined in the DTB.
`devloc_hash`	`enhf392eda5` – Ethernet device with the location hash code `f392eda5`.	`<prefix>h<hash>`: • `<prefix>` – defines the type of interface. • `<hash>` – hash code calculated from the device location string (example of a location string: `pcie:1:4:0`).	Except for USB devices.
`mac`	`enxd4570a623ebc` – Ethernet device with the MAC address `d4:57:0a:62:3e:bc`.	`<prefix>x<mac_address>`: • `<prefix>` – defines the type of interface. • `<mac_address>` – MAC address of the device, represented as a hexadecimal string without separators (colons).	Only for devices using MAC addresses for identification on the network (devices belonging to the second layer of the OSI model).
`hwaddr_hash`	`enhf392eda5` – Ethernet device with the hardware address hash code `f392eda5`.	`<prefix>h<hash>`: • `<prefix>` – defines the type of interface. • `<hash>` – hash code calculated from the hardware address of the device.	Only for ports with a hardware address.

  Prefixes that define the types of interfaces: `en` for Ethernet interfaces, `wl` for Wi-Fi interfaces, `wwan` for modem interfaces.

  Hash codes are calculated using the MurmurHash2 algorithm.

Default values for startup parameters and environment variables of VFS

For the `VfsRamFs` executable file:

```
ROOTFS = ramdisk0,0 / ext4 0
```

```
VFS_FILESYSTEM_BACKEND = server:kl.VfsRamFs
```

For the `VfsSdCardFs` executable file:

```
ROOTFS = mmc0,0 / fat32 0
VFS_FILESYSTEM_BACKEND = server:kl.VfsSdCardFs
-l nodev /tmp ramfs 0
-l nodev /var ramfs 0
```

For the `VfsNet` executable file:

```
VFS_NETWORK_BACKEND = server:kl.VfsNet
VFS_FILESYSTEM_BACKEND = server:kl.VfsNet
-l devfs /dev devfs 0
```

Mounting file systems when VFS starts

When the VfsRamFs and VfsSdCardFs programs start, only the RAMFS file system is mounted to the root directory by default. If you need to mount other file systems, this can be done not only by calling the mount() function but also by setting the startup parameters and environment variables of the VFS program.

> The `ROMFS` and `squashfs` file systems are intended for read-only operations. For this reason, you must specify the `ro` parameter to mount these file systems.

Using the startup parameter -l

One way to mount a file system is to set the startup parameter `-l <entry in fstab format>` for the VFS program.

In these examples, the devfs and ROMFS file systems will be mounted when the VFS program is started:

product_init.yaml(.in)

```
...
- name: VfsFirst
  args:
  - -l
  - devfs /dev devfs 0
  - -l
  - romfs /etc romfs ro
...
```

CMakeLists.txt

```
...
set_target_properties (${vfs_ENTITY} PROPERTIES
EXTRA_ARGS
"  - -l
  - devfs /dev devfs 0
  - -l
  - romfs /etc romfs ro")
...
```

Using the fstab file from the ROMFS image

When building a solution, you can add the `fstab` file to the ROMFS image. This file can be used to mount file systems by setting the startup parameter `-f <path to the fstab file>` for the VFS program.

In these examples, the file systems defined via the `fstab` file that was added to the ROMFS image during the solution build will be mounted when the VFS program is started:

product_init.yaml(.in)

```
...
- name: VfsSecond
  args:
  - -f
  - fstab
...
```

CMakeLists.txt

```
...
set_target_properties (${vfs_ENTITY} PROPERTIES
EXTRA_ARGS
"  - -f
  - fstab")
...
```

Using an "external" fstab file

If the `fstab` file resides in another file system instead of in the ROMFS image, you must set the following startup parameters and environment variables for the VFS program to enable use of this file:

1. `ROOTFS`. This environment variable mounts the file system containing the `fstab` file to the root directory.
2. `UNMAP_ROMFS`. If this environment variable is defined, the `fstab` file will be sought in the file system defined through the `ROOTFS` environment variable.
3. `-f`. This startup parameter is used to mount the file systems specified in the `fstab` file.

In these examples, the ext2 file system that should contain the `fstab` file at the path `/etc/fstab` will be mounted to the root directory when the VFS program starts:

product_init.yaml(.in)

```
...
- name: VfsThird
  args:
  - -f
  - /etc/fstab
  env:
    ROOTFS: ramdisk0,0 / ext2 0
    UNMAP_ROMFS: 1
...
```

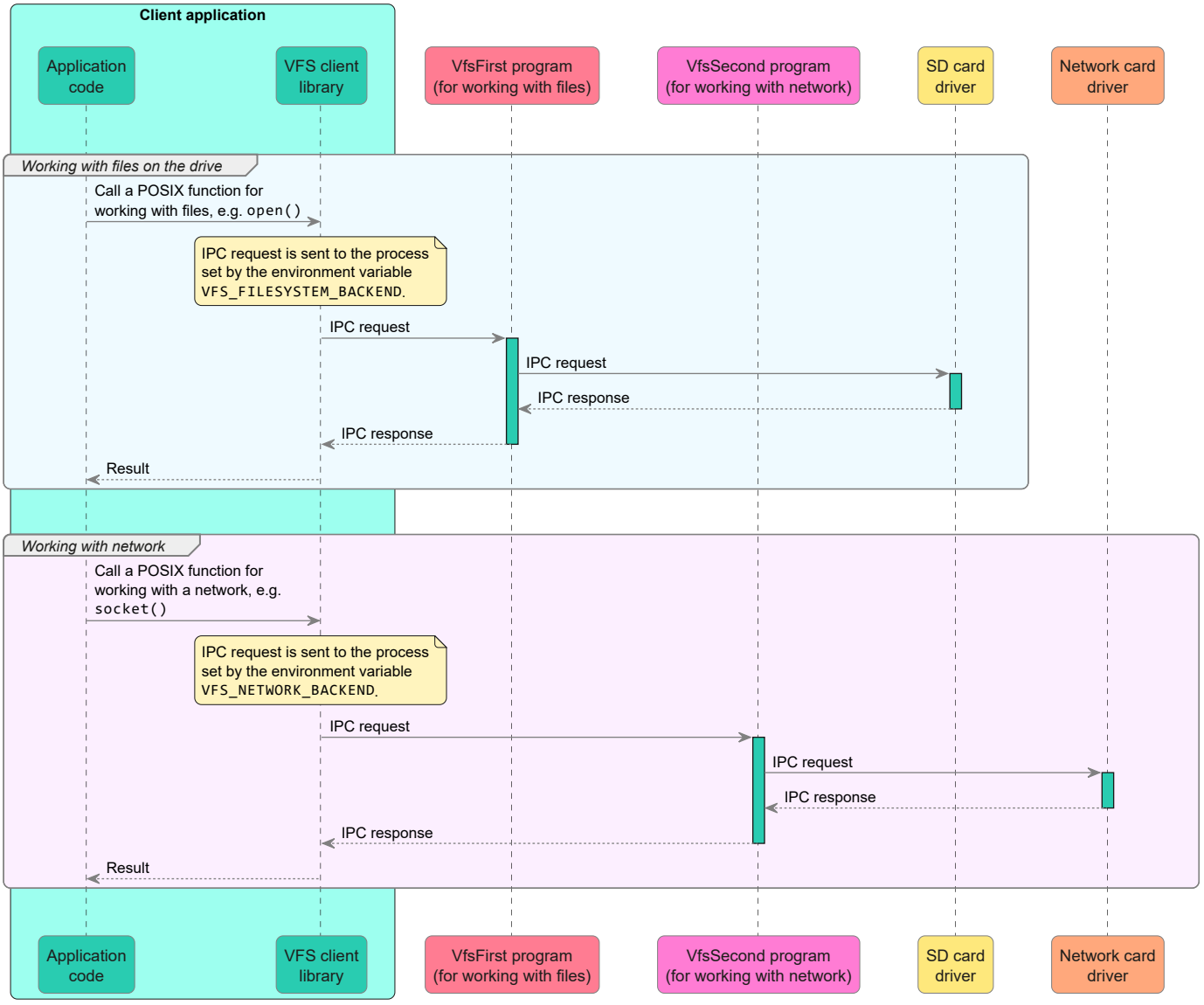
CMakeLists.txt

```
...
set_target_properties (${vfs_ENTITY} PROPERTIES
EXTRA_ARGS
"  - -f
  - /etc/fstab"
EXTRA_ENV
"  ROOTFS: ramdisk0,0 / ext2 0
  UNMAP_ROMFS: 1")
...
```

Using VFS backends to separate data streams

This example employs a secure development pattern that separates data streams related to file system use from data streams related to the use of a network stack.

The `Client` process uses file systems and the network stack. The `VfsFirst` process works with file systems, and the `VfsSecond` process provides the capability to work with the network stack. The environment variables of programs that run in the contexts of the `Client`, `VfsFirst` and `VfsSecond` processes are used to define the VFS backends that ensure the segregated use of file systems and the network stack. As a result, IPC requests of the `Client` process that are related to the use of file systems are handled by the `VfsFirst` process, and IPC requests of the `Client` process that are related to network stack use are handled by the `VfsSecond` process (see the figure below).



Separation of data streams related to file system use from data streams related to network stack use.

Init description of the example:

product_init.yaml

```yaml
entities:

- name: Client
  connections:
  - target: VfsFirst
    id: VFS1
  - target: VfsSecond
    id: VFS2
  env:
    VFS_FILESYSTEM_BACKEND: client:VFS1
    VFS_NETWORK_BACKEND: client:VFS2

- name: VfsFirst
  env:
    VFS_FILESYSTEM_BACKEND: server:VFS1
```

```
- name: VfsSecond
  env:
    VFS_NETWORK_BACKEND: server:VFS2
```

VFS relay mechanism

The VFS relay mechanism lets you implement granular distribution of data streams among multiple VFS system programs depending on the user scenarios. The VFS relay mechanism is based on VFS backends.

In contrast to creating your own VFS backend in which the rules for selecting various VFS programs for performing file or network operations must be implemented on the client application side, the VFS relay mechanism lets you use a separate VFS server program that implements such rules and several of its associated underlying VFS programs to which the VFS server program forwards requests from client applications. It is also possible to build a VFS server program that does not actually provide file system/network stack implementations itself but instead acts as a dispatcher that redirects operations on files/sockets to the underlying VFS that processes them.

The rules for selecting the underlying VFS program when creating a file or socket are generated as executable code. The user needs to implement methods that can determine whether the file or socket belongs to a specific VFS program based on the parameters of the system call. Subsequent operations performed on the created files or sockets are automatically redirected to the VFS program that was assigned to the file or socket when it was created. For more details, refer to "Performing file and network operations" below.

The VFS component comes with the `vfs::relay` library, which lets you build a VFS server program that will redirect calls of functions for working with files and the network to VFS client programs. Alternatively, the `vfs::relay` library can be used locally, in which case the `vfs::relay` library is linked directly to the client application. For more details, refer to "VFS relay in a server application" and "VFS relay in a client application" below.

The public methods of the VFS relay backend are described in the header file `sysroot-*-kos/include/vfs/relay.h`.

See also vfs_relay example located in `examples/vfs_relay` from the SDK.

VFS relay in a server application

The `vfs::relay` library can be linked to a user program performing the VFS server role. This usage case allows the separation of data streams on the server side without the need to rebuild or change the configuration of client applications. In this case, the operation can be performed both in the VFS server program itself and it can be redirected to the underlying VFS client programs via IPC.

In this mode, a local opening context and a local handle (`Handle`) will be generated for files and sockets opened in the VFS server program (for example, `open()` in the server code or the libraries that it uses), and this handle can only be used in the context of this server program and cannot be transmitted over IPC to VFS client programs.

For files and sockets that were created when executing IPC requests from client applications, an OCAP badge is generated and this will create a unique OCAP handle that indicates the context of opening a file or socket. This handle can be passed to client applications and can be passed via IPC to the underlying VFS programs connected to the same VFS server.

For more details, refer to Managing handles.

Using the VFS relay backend in a server application

To build a program that performs the VFS server role using the VFS relay backend, you must specify the dependency on the `vfs::server` and `vfs:relay` libraries.

Example of specifying libraries for a program that performs the VFS server role and uses a VFS relay backend with client and local backends:

```
find_package (vfs REQUIRED)
target_link_libraries (${PROJECT_NAME} vfs::server vfs::relay vfs::client vfs::local)
```

The startup of a program that performs the VFS server function is configured in `product_init.yaml` by specifying the `VFS_FILESYSTEM_BACKEND` and/or `VFS_NETWORK_BACKEND` environment variables.

The default VFS program is specified after the name of the relay backend, separated by a colon. If you do not specify a relay backend, the relay mechanism will not work, the server will be equivalent to a normal one, and function calls will not be redirected.

Before specifying the VFS relay backend, you must specify that the top-level backend is the `server`, and also specify the server backend ID, such as `server:RelayVfs:relay:client:kl.VfsEntity`.

To specify the underlying VFS programs that will be automatically connected to the VFS relay at startup, you must use the `VFS_RELAY_DESCENDANTS` environment variable. The entry format for the child client is `Backend_label=backend_configuration`. Based on the backend label being used, you can obtain a pointer to the VFS structure when redefining the VFS relay methods (see below). In `VFS_RELAY_DESCENDANTS`, you can simultaneously list multiple child backends separated by the `;` symbol.

Example `product_init.yaml` configuration for the `RelayVfsServer` program. It uses the VFS relay backend to which the `kl.VfsEntity` child program is connected as the default backend. The local backend and client backend with the labels `LocalVfs` and `UserVfsServer` are also connected. Network and file operations are handled through a shared VFS relay backend.

```
- name: example.RelayVfsServer
  path: RelayVfsServer
  env:
    VFS_FILESYSTEM_BACKEND: "server:RelayVfs:relay:client:kl.VfsEntity"
    VFS_NETWORK_BACKEND: "server:RelayVfs:relay:client:kl.VfsEntity"
    VFS_RELAY_DESCENDANTS: "LocalVfs=local;UserVfsServer=client:task:example.UserVfsServer"
@INIT_RelayVfsServer_ENTITY_CONNECTIONS@
```

If the VFS server itself implements any file system/network stack available through the `local` backend and you need to use this particular implementation in the default relay mechanism, the initialization in `product_init.yaml` should be as follows:

```
- name: example.RelayVfsServer
  path: RelayVfsServer
  env:
    VFS_FILESYSTEM_BACKEND: "server:RelayVfs:relay:local:"
```

```
    VFS_NETWORK_BACKEND:    "server:RelayVfs:relay::relay:local:"
    VFS_RELAY_DESCENDANTS: "UserVfsServer=client:task:example.UserVfsServer;VfsFsServer=client:task:kl.VfsEntity"
```

> Important!
> In the `product_init.yaml` configuration for the server application, the values specified in `VFS_FILESYSTEM_BACKEND` and `VFS_NETWORK_BACKEND` must match.

VFS relay in a client application

The `vfs::relay` library can be linked to a user program. Use of the VFS relay as a backend for a client program allows the program to simultaneously use the local backend to perform operations on some files or sockets and use the client backend to perform operations on individual VFS programs. In this mode, for each opened file/socket of the VFS relay, the backend stores the local file opening context that contains information about the underlying backend being used. A pointer to the opening context is a local handle (`handle`), which can only be used in the context of this program and cannot be passed to other programs via IPC.

Using the VFS relay backend in a client application

To build a client program with support for the VFS relay backend, you must specify the dependency on the `vfs:relay` library in the `CMake` file for building this program.

Example of specifying libraries when using the VFS relay backend with the underlying local and client backends and the libraries required for the local backend:

```
find_package (vfs REQUIRED)
target_link_libraries (${PROJECT_NAME} vfs::relay vfs::client vfs::local vfs::lib_fs)
```

In the startup description for the client program in the `product_init.yaml` file, you must specify the use of the VFS relay backend and the backend that will be used as the default VFS via the `VFS_FILESYSTEM_BACKEND` and/or `VFS_NETWORK_BACKEND` environment variables.

The default VFS program is specified after the name of the VFS relay backend, separated by a colon.

To specify the underlying VFS programs that will be automatically connected to the VFS relay at startup, you must use the `VFS_RELAY_DESCENDANTS` environment variable. The entry format for the child client is `Backend_label=backend_configuration`. Based on the backend label being used, you can obtain a pointer to the VFS structure when redefining the VFS relay methods (see below). In `VFS_RELAY_DESCENDANTS`, you can simultaneously list multiple child backends separated by the `;` symbol.

Example `product_init.yaml` configuration for the `SimpleMultiVfs` program. It uses the VFS relay backend to which the `kl.VfsEntity` child program is connected as the default backend, and the local backend is additionally connected. Network and file operations are handled through a shared VFS relay backend.

```
- name: example.SimpleMultiVfs
  path: SimpleMultiVfs
  env:
    VFS_FILESYSTEM_BACKEND: "relay:client:kl.VfsEntity"
    VFS_NETWORK_BACKEND: "relay:client:kl.VfsEntity"
```

```
    VFS_RELAY_DESCENDANTS: "LocalVfs=local"
@INIT_SimpleMultiVfs_ENTITY_CONNECTIONS@
```

Performing file and network operations

When completing user requests, the main task of the VFS relay backend is to select the correct underlying backend for redirecting the operation. The underlying backend selection logic is implemented by the backend user in the form of executable code that is registered as a callback function. For each type of resource (file, socket, pipe), you can set an independent callback function in which the logic for selecting the underlying backend will be implemented. Operations performed on an already created resource (file, socket, pipe) will be automatically redirected to the VFS backend that was used when creating the file.

The functions for registering callback functions are defined in the header file `sysroot-*-kos/include/vfs/relay.h`.

Creating a file and performing file operations

The callback function for file creation is registered using the `relay_set_file_cb()` function.

The parameters received by the callback function are the client connection context and the path to the file for which the operation is performed. The path is transmitted in absolute format, and the size of the `path` array is equal to `MAX_PATH`.

Socket creation

The callback function for socket creation is registered using the `relay_set_socket_cb()` function.

The parameters received by the callback function are the client connection context and the set of attributes of the created socket (domain, type, protocol).

Creating channels

The callback function for creating an unnamed channel (pipe) is registered using the `relay_set_pipe_cb()` function.

Auxiliary functions

The pointer to the underlying backends connected to the VFS relay can be received by using the `relay_get_descendant()`, `relay_get_fs_vfs()`, `relay_get_net_vfs`, and `relay_get_default_vfs()` functions.

Set of template callback functions

A library containing a set of methods that can be registered as callback functions when performing operations in the header file `sysroot-*-kos/include/vfs/relay_cb.h` is supplied together with the VFS relay backend.

Considerations for working with file systems

The functionality for working with file systems is available to applications via POSIX functions and other functions of the standard C library.

The support limitations and special considerations for POSIX implementation are described in the following sections:

- POSIX support limitations.
- POSIX implementation specifics.

In KasperskyOS, the implementation of certain functions of the standard C library for file system management differs from the implementation of these functions in Linux and other UNIX-like operating systems. Information about these functions is provided in the table below.

Description and implementation details of the functions of the standard C library for file system management

Function	Purpose	Implementation specifics	Header file
`mount()`	Mount a file system.	When starting the VFS system program, RAMFS is automatically mounted as the '`/`' root file system. After startup of VFS, a single remounting of the root file system to any other file system is allowed if there are no open files in RAMFS. If this remounting occurs, the contents of RAMFS will be freed. Due to the fact that initialization of the block level occurs asynchronously with VFS operation, prior to mounting a block device you must verify its readiness. When attempting to mount a block device that has not yet been registered in the block device driver, the `mount()` function returns the `ENODEV` error. You can use the virtual file system `devfs` to get a list of devices that are available to be mounted. To do so, you must mount `devfs` to the `/dev` directory and print the list of available devices by using `readdir()` in the `devfs` mount directory. For more details, refer to File system mounting functions	`sys/mount.h`
`umount()`	Unmount the file system.	When executing the `umount()` function, VFS verifies that there are no open files (file handles) in the subtree of the file system being unmounted. File closing events may reach VFS asynchronously. Therefore, when there are open files in the file system being unmounted, the VFS performs repeated unmount attempts during the timeout for attempts (1 second by default), after which it returns the `EBUSY` error to the calling process. For more details, refer to File system mounting functions	`sys/mount.h`
`sync()`	Write the file system caches to disk.	The function was not implemented. When calling `sync()`, nothing happens because POSIX does not allow a `sync()` call to return an error.	`unistd.h`
`fsync()`	Synchronize changes in the file with the disk device.	It works if the call is supported in the file system in which the file is open, otherwise it returns the `ENOSYS` error.	`unistd.h`
`fdatasync()`	Synchronize changes in the file with the disk device.	It works if the call is supported in the file system in which the file is open, otherwise it returns the `ENOSYS` error.	`unistd.h`

Function	Purpose	Implementation specifics	Header file
`open()`	Open a file.	It is not permissible to use the `O_CREAT` flag without explicitly specifying the `mode` parameter.	`fcntl.h`
`timerfd_create()`	Create a timer that informs about the onset of an event via file handle.	Only the following `clockid` parameter values are supported: • `CLOCK_MONOTONIC` • `CLOCK_REALTIME`	`sys/timerfd.h`
`timerfd_settime()`	Start, stop, or configure a timer created by the `timerfd_create()` function.	You cannot change the system time for timers with the `clockid` parameter set to `CLOCK_REALTIME`. The `TFD_TIMER_CANCEL_ON_SET` and `TFD_IOC_SET_TICKS` flags are not defined or supported.	`sys/timerfd.h`

Supported file systems

The SDK provides the following file system drivers:

- Proprietary implementation of the VFAT file system driver (the `fat12`, `fat16`, `fat32`, and `vfat` file systems are supported).
- Proprietary implementation of the ext2/3/4 file system driver based on the open-source lwext4 implementation (the `ext`, `ext2`, `ext3`, and `ext4` file systems are supported).
- Proprietary implementation of the driver for the `virtiofs` file system intended for sharing files between the host system and virtual machines.

VFS executable files

The `precompiled_vfs` CMake package contains the following executable files:

- `VfsRamFs`
- `VfsSdCardFs`
- `VfsNet`

The `VfsRamFs` and `VfsSdCardFs` executable files include the `vfat` and `lwext4` libraries. The executable file `VfsNet` does not include implementations of file system drivers.

Including drivers in a VFS program build

When building a solution, you can build a modified VFS program by adding the required file system drivers. To do so, use the following `CMake` commands:

```
# CMake package of the VFAT file system driver
find_package (VFAT REQUIRED)
include_directories (${VFAT_INCLUDE})

# CMake package of the ext4 file system driver
find_package (LWEXT4 REQUIRED)
include_directories (${LWEXT4_INCLUDE})

# Linking the libraries of file system drivers into VFS.
```

```
target_link_libraries (VfsFs
                       fs::vfat
                       fs::lwext4)
```

For more details, refer to Building a VFS executable file.

Mounting file systems

Details:

- Mounting file systems when VFS starts
- File system mounting functions

File system mounting functions

In KasperskyOS, operations with file systems and the network are executed via a separate system program that implements a virtual file system (VFS).

mount() function

The `mount()` function is intended for mounting file systems. When using this function in KasperskyOS, it is important to consider the following details regarding its implementation:

- Remounting of the root file system is prohibited in KasperskyOS. The root file system must always be mounted through the `ROOTFS` environment variable that is passed to the VFS program and contains a single string in `fstab` format. For an example of using the ROOTFS environment variable, see Mounting file systems when VFS starts.
- To specify the types of file systems, the additional value `auto` is available for mounting a file system that supports automatic mounting.
- The number of mountable file systems and supported mount flags in KasperskyOS is limited compared to the implementation of the function in the `libc` library.
- The mount source cannot be a directory.

By default, the VFS program is not mapped to a program implementing a block device. To bind a process of the utilized virtual file system with the required block device, use the `blkdev::entity_REPLACEMENT` property of the `set_target_properties` function in the `CMakeLists.txt` file for building the `Einit` program. Possible values for the `blkdev::entity_REPLACEMENT` property (implementations of block devices):

- `ramdisk::entity` – RAM disk that reads the `ramdisk<N>.img` files from ROMFS and provides them as the `ramdisk<N>` file systems, where `N` is a positive integer.
- `sdcard::entity_hw` – SD card driver (MMC).
- `usb_storage_entity::entity` – USB Mass Storage driver (for hardware platforms).
- `""` (empty string) – interaction between the block device and the virtual file system is not required.

The `mount()` function is declared in the header file `sysroot-*-kos/include/sys/mount.h` from the KasperskyOS SDK:

```
int mount(const char *source, const char *target,
          const char *filesystemtype, unsigned long mountflags,
          const void *data);
```

Function parameters:

- `source`

  Mount source: device or virtual object (dummy). Supported formats:

  - `<device_name>[_p<volume_number>]` – full name of the device.
  - `PARTLABEL=<label>`, where `label` refers to the GPT (Globally Unique Identifier Partition Table) partition label.
  - `PARTUUID=<uuid>`, where `uuid` refers to the GPT partition GUID.

  When mounting file systems that do not require a physical device (for example, ROMFS, ramfs, devfs), the `source` parameter is ignored.

- `target`

  Directory in which the file system will be mounted. This directory must be created before the `mount()` function is called.

- `filesystemtype`

  Type of file system. Permissible values:

  - `romfs` – read-only file system.
  - `ramfs` – file system that runs in RAM. Used by default for the root file system.
  - `devfs` – virtual file system for device management. Supported devices:
    - `log` – equivalent of `/dev/log` for system logging.
    - `null` – equivalent of `/dev/null` for redirecting and ignoring output.
    - `urandom` – equivalent of `/dev/random` for generating pseudorandom numbers.
    - `bpf`, `npf`, `tun0/1/2/3` – network devices for working with network interfaces and filters.
  - `fat12`, `fat16`, `fat32`, `vfat` – file systems of the FAT family. One implementation for the entire family.
  - `ext`, `ext2`, `ext3`, `ext4` – file systems of the ext family. One implementation for the entire family.
  - `ffs` – file system used in NetBSD.

- `virtiofs` – file system for shared access between the guest KasperskyOS running in QEMU and the host OS.
- `auto` – automatic mounting of the first file system from the list of registered file systems that satisfies the following conditions: requires a device and may be mounted automatically (for example, `fat32`, `ext` and `ffs`).

- `mountflags`

  Mount flags. Available values:

  - `MS_RDONLY` – mount the file system in read-only mode.
  - `MS_STRICTATIME` – always update the last access time of files when their contents are changed.
  - `MS_SYNCHRONOUS` – synchronous write to the file system.

- `data`

  Additional data for mountable file systems. Empty string, or `NULL` if the parameter does not have to be specified.

  The following values are permissible for the file systems of the ext family:

  - `expand` – expand the file system over the entire available space of the volume if this is permitted by the partition structure.
  - `check` – initiate full auto-recovery of the file system when mounting.

  The following values are permissible for the `virtiofs` file system:

  - `tag=<...>` – label of the `virtiofs` file system to be mounted. It is indicated in the QEMU parameters at startup. This parameter is required when mounting virtiofs

Returns `0` if successful, otherwise returns `-1`. If an error occurs, the `errno` variable is set to a value indicating the cause of the error. The possible values of `errno` match the standard values defined in the `libc` library.

umount() function

This function is intended for unmounting file systems and is declared in the header file `sysroot-*-kos/include/sys/mount.h` from the KasperskyOS SDK:

```
int umount(const char *target);
```

Function parameters:

- `target`

  Directory from which the file system will be unmounted.

Returns `0` if successful, otherwise returns `-1`. If an error occurs, the `errno` variable is set to a value indicating the cause of the error. The possible values of `errno` match the standard values defined in the `libc` library.

Example function call:

```
#include <sys/mount.h>

int main(void)
{
    // Create directory /c and specify
    // the access rights for all users
if (mkdir("/c", S_IRWXU | S_IRWXG | S_IRWXO)) {
        fprintf(stderr, "Failed to create dir\n");
        return -1;
    }

    // Mount the ext2 file system from block device mmc0_p0
    // to directory /c using standard mount flags
    // and without additional data
    if (mount("mmc0_p0", "/c", "ext2", 0, "")) {
        fprintf(stderr, "Failed to mount 'ext2'\n");
        return -1;
    }

    // Create directory /c/ram and specify
    // the access rights for all users
    if (mkdir("/c/ram", S_IRWXU | S_IRWXG | S_IRWXO) != 0)
    {
        printf("Failed to create \"/c/ram\" dir\n");
        return EXIT_FAILURE;
    }

    // Mount the ramfs file system to directory /c/ram
    // using standard mount flags
    // and without additional data
    if (mount("nodev", "/c/ram", "ramfs", 0, "") != 0)
    {
        printf("Failed to mount ramfs, /c/ram (error %d: \"%s\")\n",
               errno,
               strerror(errno));
        return EXIT_FAILURE;
    }

    // ...

    // Unmount the file system from directory /c/ram
    if (umount("/c/ram")) {
        fprintf(stderr, "Failed to umount 'ramfs'\n");
        return -1;
    }

    return 0;
}
```

For other ways to mount file systems, see the section titled Mounting file systems when VFS starts.

Notification subscription functions

The notifications API provides a mechanism for subscribing to file system events. This mechanism allows applications to track changes made to files and directories, such as object access, modified contents, and file creation or deletion.

The API is defined in the header file `sysroot-*-kos/include/kos/subscribe.h` from the KasperskyOS SDK.

Using the API

To use the notification mechanism, you must do the following:

1. Create a subscription object by using the `subscribe_create()` function.
2. Add relevant events by using the `subscribe_ctl()` function.
3. Subscribe to the delivery of event notifications by using the `subscribe_wait()` function.
4. If necessary, complete the delivery of notifications by using one of the `subscribe_drop_and_wake()` or `subscribe_free()` functions.

Information about API functions

Subscribe.h functions

Function	Information about the function
`subscribe_create()`	Purpose Allocates and initializes the subscription object. Parameters N/A Returned values: Pointer to the subscription object in case of success, `NULL` or an error number in case of failure.

`subscribe_ctl()`	Purpose
	Adds or removes an event entry to/from the subscription object monitoring list.
	Parameters
	• `[in] self` – subscription object.
	• `[in] op` – type of operation:
	  • `SUBSCRIBE_CTL_ADD = 1` – adds an event entry to the monitoring list.
	  • `SUBSCRIBE_CTL_DEL` – deletes all events associated with the specified file handle.
	• `[in] event` – event structure.
	Returned values:
	0 in case of success, -1 and an error code in case of failure.

`subscribe_wait()`	Purpose Waits for events from the list associated with a subscription object. Parameters • `[in] self` – subscription object. • `[out] events` – pointer to the array containing received events. • `[in] max_events` – size of the `events` array. • `[in] timeout` – time to wait in milliseconds (-1 for an infinite timeout). Returned values: Number of received events in case of success, -1 and an error code in case of failure.
`subscribe_drop_and_wake()`	Purpose Clears the monitoring list and wakes up all threads that are waiting for events on this subscription object. Parameters • `[in] self` – subscription object. Returned values: 0 in case of success, -1 and an error code in case of failure.

`subscribe_free()`	Purpose Clears the monitoring list, wakes all threads that are waiting for events on this subscription object, and destroys the subscription object. Parameters • `[in] self` – subscription object. Returned values: 0 in case of success, -1 and an error code in case of failure.

Event structure and event masks

The `subscribe_event` structure contains information about an event:

- `fd` – handle of the object to which the event belongs.
- `event_mask` – mask of events that were registered:
  - `N_ACCESS (1 << 0)` – access was granted to the object.
  - `N_MODIFY (1 << 1)` – the object or its contents have been modified.
  - `N_ATTRIB (1 << 2)` – the metadata of the object or its contents have been modified.
  - `N_CLOSE (1 << 3)` – the object or its contents have been closed.
  - `N_OPEN (1 << 4)` – the object or its contents have been opened.
  - `N_CREATE (1 << 5)` – a file or directory was created inside the tracked object.
  - `N_UNLINK (1 << 6)` – the object or its contents have been deleted.
  - `SUBSCRIBE_MASK_ALL` – used to subscribe to all events.

Function for tracking connected drives (kos_devfs.h)

The API is defined in the header file `sysroot-*-kos/include/kos_devfs.h` from the KasperskyOS SDK.

The `wait_for_dev()` function allows programs to use the notification mechanism instead of constantly polling the file system to track new devices appearing in the system. The function receives a pointer to the name of the target device and the time to wait, in seconds. It returns `true` if the expected device was connected earlier or during the wait timer, and returns `false` if the wait timed out or an error occurred.

The `wait_for_dev()` function also uses the auxiliary function `normalize_blkdevname()`, which converts the device name from comma format to underscore format. For example, `Ahci0Port0,0` is converted to `Ahci0Port0_p0`.

The `WAIT_FOR_DEV_MAX_DEV_NAME_SIZE` and `WAIT_FOR_DEV_DEFAULT_TIMEOUT_SEC` constants limit the maximum device name length to 128 characters and set the default timeout value to 10 seconds.

How it works

The `wait_for_dev()` function performs the following actions:

1. Convert the device name from comma format to underscore format using the `normalize_blkdevname()` function.
2. Check for the presence of a device in the system immediately after startup. If the device is present in the system, the function returns `true` and exits.
3. Create the `/dev` directory and mount `/devfs` as necessary.
4. Configure notifications about file system events:
    a. Using the `subscribe_create()` function to create a subscription object.
    b. Open the `/dev` directory for monitoring.
    c. Use the `subscribe_ctl()` function to register a subscription to an object creation event.
5. Use `subscribe_wait()` to wait for the relevant device creation event
6. Use the `subscribe_free()` function to release resources, close the handles, and return `true` if the expected device was connected; return `false` if the expected device has timed out or an error has occurred.

Example use of the API function:

```
#include <assert.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mount.h>

#define _VFS_SERVER_INTERFACE
#include <vfs/basic.h>
#include <kos_devfs.h>

static _vfs_server_thread_routine_t thread_routine;
static void *thread_routine_arg;

void _vfs_server_set_main_thread_routine(_vfs_server_thread_routine_t func, void *arg)
{
    thread_routine = func;
    thread_routine_arg = arg;
}

int main(void)
{
    char *devname = BLKDEV0;
    char *dirname = "/mnt1";
    char *fsname = "fat32";
    long unsigned int flags = 0;
    char *options = "";

    if (!wait_for_dev(devname, WAIT_FOR_DEV_DEFAULT_TIMEOUT_SEC))
    {
        fprintf(stderr, BLKDEV0" is missed\n");
        return false;
    }

    if (mkdir(dirname, S_IRWXU | S_IRWXG | S_IRWXO) != 0 && errno != EEXIST)
    {
        perror("Failed to mkdir");
        return false;
    }
```

```
    if (mount(devname, dirname, fsname, flags, options) != 0)
    {
        perror("Failed to mount fat32");
        return false;
    }

    assert(thread_routine != NULL);

    (*thread_routine)(thread_routine_arg);
    return EXIT_SUCCESS;
}
```

Considerations for working with a network

This section provides information about working with a network for KasperskyOS development.

Configuration and status of network interfaces (kos_net.h)

The API is defined in the header file `sysroot-*-kos/include/kos_net.h` from the KasperskyOS SDK.

The API lets you do the following:

- Get a list of network interfaces available in the system.
- Send data on available network interfaces to standard output.
- Convert a textual representation of an IPv6 address into binary format.
- Wait for the appearance of network interfaces with the defined parameters in the list of available network interfaces.
- Configure the available network interfaces with IPv4 and IPv6 addressing.

Information about API functions is provided in the table below.

To send information about available network interfaces to standard output, use the `list_network_ifaces()` function. This function does not have any parameters. The standard output displays the following data about each network interface: name, type, activity status and address of the network interface with its subnet mask.

Example output of the `list_network_ifaces()` function:

```
Name    Type    Running    Address/mask
lo0     inet    Yes        ::1/ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff
lo0     inet    Yes        fe80:1::1/ffff:ffff:ffff:ffff::
lo0     inet    Yes        127.0.0.1/255.0.0.0
en0     mac     Yes        52:58:00:12:34:bb
en0     inet    Yes        10.0.2.15/255.255.255.
en0     inet    Yes        fe80:2::5058:ff:fe12:34bb/ffff:ffff:ffff:ffff::
```

To convert the textual representation of an IPv6 address into binary format, use the `ip6_parse_addr()` function. This function uses the `ip_str` input parameter to receive the textual representation of an IPv6 address in accordance with paragraphs 2.2 and 2.3 of the RFC 4291 specification in the following format:

`<ipv6_address>[/<prefix_size>]`

The function uses the `dst_mask` output parameter to return the pointer to the `sockaddr_in6` structure, which contains the prefix mask in the `sin6_addr` field. The function uses the `dst_ip` output parameter to return the pointer to the other `sockaddr_in6` structure, which contains the IPv6 address in the `sin6_addr` field. The `sockaddr_in6` structure is defined in the header file `sysroot-*-kos/include/strict/posix/netinet/in.6`.

Use the `wait_for_iface()` function to wait for the appearance of a network interface with the necessary parameters in the list of available network interfaces. This function uses the `name` input parameter to receive the name of the network interface for which the function is waiting. If this parameter is not specified (`NULL`), the function will wait for the appearance of any network interface except those whose names begin with `lo*`. This function uses the `wait_flags` input parameter to receive the flags that determine the wait parameters – availability of IP addresses and default gateways. These flags can be combined. The timeout for waiting for the necessary network interface (in seconds) is defined in the `timeout` parameter.

To configure the available network interfaces using IPv4 and IPv6 addressing, the API provides the following two functions: `configure_net_iface()` and `configure_net_iface6()`. To simplify configuration of network interfaces, there are macros that you can use as input parameters for these functions (see the table below).

Macros of the configure_net_iface() and configure_net_iface6() functions

Macro	Description	Default value	Use in `configure_net_iface()`	Use in `configure_net_iface6()`
`DEFAULT_INTERFACE`	Name of the network interface used for the connection	`en0`	`Yes`	`Yes`
`DEFAULT_ADDR`	IP address assigned to the network interface	`10.0.2.15`	`Yes`	`No`
`DEFAULT_MASK`	Subnet mask	`255.255.255.0`	`Yes`	`No`
`DEFAULT_GATEWAY`	Default gateway	`10.0.2.2`	`Yes`	`No`
`DEFAULT_MTU`	Maximum size of a transmitted packet, in bytes (Maximum Transmission Unit, MTU)	`1500`	`Yes`	`No`
`DEFAULT_TIMEOUT`	Timeout (in seconds)	`60`	`Yes`	`Yes`

Information about API functions

kos_net.h functions

Function	Information about the function
`net_iface_exist()`	Purpose Checks the availability of a network interface. Parameters • [in] `ifname` – name of the network interface to check. Returned values: • `1` – the network interface exists. • `0` – the network interface does not exist. • `-1` – an error occurred

Function	Information about the function
`configure_net_iface()`	Purpose Configures a network interface using IPv4 addressing. The `ip`, `mask`, and `gw` input parameters must be specified in the format `X.X.X.X`, where `X` is a decimal number from 0 to 255. Parameters • [in] `name` – name of the network interface that needs to be configured. • [in] `ip` – IPv4 address that will be assigned to the network interface. • [in] `mask` – subnet mask. • [in] `gw` – IPv4 address of the default gateway. • [in] `mtu` – maximum size (in bytes) of a packet. Returned values: Returns `true` if successful, otherwise returns `false`.
`ip6_parse_addr()`	Purpose Converts a textual representation of an IPv6 address into binary format, and (optionally) generates a prefix mask based on the specified prefix size. Parameters • [in] `ip_str` – textual representation of the IPv6 address. • [out] `dst_mask` – pointer to the `sockaddr_in6` structure containing the prefix mask in the `sin6_addr` field. • [out] `dst_ip` – pointer to the `sockaddr_in6` structure containing the IPv6 address in the `sin6_addr` field. Returned values: Returns `true` if successful, otherwise returns `false`.
`configure_net_iface6()`	Purpose Configures a network interface using IPv6 addressing. Parameters • [in] `name` – name of the network interface that needs to be configured. [in] `ip` – IPv6 address according to paragraphs 2.2 and 2.3 of the RFC 4291 specification. • [in] `gw` – IPv6 address of the default gateway according to paragraph 2.2 of the RFC 4291 specification, or `NULL` if you do not need to specify this parameter. Returned values: Returns `true` if successful, otherwise returns `false`.
`list_network_ifaces()`	Purpose Gets a list of network interfaces that are available in the system and sends data on each network interface to standard output. Parameters N/A Returned values Returns `true` if successful, otherwise returns `false`.

Function	Information about the function

`wait_for_iface()`	Purpose
	
	Waits for the appearance of an available network interface that meets the requirements defined by the wait flags.
	
	Parameters
	
	• [in] `name` – name of the network interface for which the function is waiting. If this parameter is not specified (`NULL`), the function waits for the appearance of any network interface except those whose names begin with `lo*`.
	
	• [in] `wait_flags` – wait flags:
	
	  • IWF_EXISTS – wait for the appearance of a network interface in the list of available network interfaces (always used by default). In this case, the network interface does not have to be active.
	
	  • IWF_IP4 – wait for the assignment of an IPv4 address to the network interface.
	
	  • IWF_IP6 – wait for the assignment of an IPv6 address to the network interface.
	
	  • IWF_GW4 – wait for the appearance of an entry for an IPv4 gateway in the routing table. The entry may be added automatically (via DHCP) or manually by a user.
	
	  • IWF_GW6 – wait for the appearance of an entry for an IPv6 gateway in the routing table. The entry may be added automatically (via DHCP) or manually by a user.
	
	• [in] `timeout` – timeout (in seconds) to wait for the network interface with the specified parameters.
	
	Returned values:
	
	Returns `true` if successful, otherwise returns `false`.

Example use of API functions:

```
#include <kos_net.h>

int main(int argc, char *argv[])
{
    // Wait 60 seconds for the network interface named en0 to appear in the list of available network interfaces
    if (!wait_for_iface(DEFAULT_INTERFACE, IWF_EXISTS, DEFAULT_TIMEOUT))
    {
        perror("Network interface does not exist");
        return EXIT_FAILURE;
    }en0


    // Configure network interface en0 with the default values
    if (!configure_net_iface(DEFAULT_INTERFACE, DEFAULT_ADDR, DEFAULT_MASK, DEFAULT_GATEWAY, DEFAULT_MTU))
    {
        perror("Can't configure network");
        return EXIT_FAILURE;
    }
}
```

Vector network functions sendvec() and recvvec()

The API is defined in the header file `sysroot-*-kos/include/kos/vec_socket.h` from the KasperskyOS SDK.

The API lets you send multiple network packets in a single IPC message, which speeds up network operations in KasperskyOS.

When using the `send()` or `recv()` functions of the standard C library, a separate IPC call is created for each call of these functions, which may slow down network operations. Due to the design of the KasperskyOS architecture, sending a large number of small packets results in a large number of IPC messages.

In contrast to the `readv()`/`writev()` functions of the standard C library, the `sendvec()`/`recvvec()` interface supports not only forwarding/receiving multiple packets at one socket but also forwarding/receiving multiple packets at multiple sockets.

Information about API functions is provided in the table below.

Using the API

To send packets, you must initialize the network interface, create sockets for connecting to the server, create and initialize the `outvec` structure containing the handles of sockets for connecting to the server and the pointers to packets to be sent, and then call the `sendvec()` function.

To receive packets, you must create sockets for receiving incoming connection requests, create and initialize the `invec` structure containing the handles of sockets for receiving packets and the pointers to buffers used to receive packets, accept incoming connection requests, and then call the `recvvec()` function.

Data structures for API are defined in the header file `sysroot-*-kos/include/kos/vec_types.h` from the KasperskyOS SDK.

Information about API functions

vec_socket.h functions

Function	Information about the function
`sendvec()`	Purpose Sends multiple packets to multiple sockets in a single IPC message. Parameters • `[in,out] vec` – pointer to the array containing the handles of sockets used to connect with the server and the pointers to the packets to be sent. The transmission results for each element (number of sent bytes or error) are also written to this array. The structure is defined in the header file `sysroot-*-kos/include/kos/vec_types.h` from the KasperskyOS SDK. • `[in] count` – number of elements in the `vec` parameter. Returned values: If successful, the function returns the number of successfully sent packets, otherwise it returns an error code.
`recvvec()`	Purpose Receives multiple packets from multiple sockets. Parameters • `[in,out] vec` – pointer to the array containing the handles of sockets for receiving packets and the pointers to buffers used to receive packets. The receipt results for each element (number of received bytes or error) are also written to this array. The structure is defined in the header file `sysroot-*-kos/include/kos/vec_types.h` from the KasperskyOS SDK. • `[in] count` – number of elements in the `vec` parameter. Returned values: If successful, the function returns the number of successfully received packets, otherwise it returns an error code.

Dynamically configuring the network stack

To change the default network stack parameters, use the `sysctl()` function or `sysctlbyname()` function that are declared in the header file `sysroot-*-kos/include/sys/sysctl.h` from the KasperskyOS SDK. The parameters that can be changed are presented in the table below.

Configurable network stack parameters

Parameter name	Parameter description
`net.inet.ip.ttl`	Maximum time to live (TTL) of sent IP packets. It does not affect the ICMP protocol.
`net.inet.ip.mtudisc`	If its value is set to `1`, "Path MTU Discovery" (RFC 1191) mode is enabled. This mode affects the maximum size of a TCP segment (Maximum Segment Size, or MSS). In this mode, the MSS value is determined by the limitations of network nodes. If "Path MTU Discovery" mode is not enabled, the MSS value does not exceed the value defined by the `net.inet.tcp.mssdflt` parameter.
`net.inet.tcp.mssdflt`	MSS value (in bytes) that is applied if only the communicating side failed to provide this value when opening the TCP connection, or if "Path MTU Discovery" mode (RFC 1191) is not enabled. This MSS value is also forwarded to the communicating side when opening a TCP connection.
`net.inet.tcp.minmss`	Minimum MSS value, in bytes.
`net.inet.tcp.mss_ifmtu`	If its value is set to `1`, the MSS value is calculated for an opened TCP connection based on the maximum size of a transmitted data block (Maximum Transmission Unit, or MTU) of the employed network interface. If its value is set to `0`, the MSS value for an opened TCP connection is calculated based on the MTU of the network interface that has the highest value for this parameter among all available network interfaces (except the loopback interface).
`net.inet.tcp.keepcnt`	Number of times to send test messages (Keep-Alive Probes, or KA) without receiving a response before the TCP connection will be considered closed. If its value is set to `0`, the number of sent keep-alive probes is unlimited.
`net.inet.tcp.keepidle`	TCP connection idle period, after which keep-alive probes begin. This is defined in conditional units, which can be converted into seconds via division by the `net.inet.tcp.slowhz` parameter value.
`net.inet.tcp.keepintvl`	Time interval between recurring keep-alive probes when no response is received. This is defined in conditional units, which can be converted into seconds via division by the `net.inet.tcp.slowhz` parameter value.
`net.inet.tcp.recvspace`	Size of the buffer (in bytes) for data received over the TCP protocol.
`net.inet.tcp.sendspace`	Size of the buffer (in bytes) for data sent over the TCP protocol.
`net.inet.udp.recvspace`	Size of the buffer (in bytes) for data received over the UDP protocol.
`net.inet.udp.sendspace`	Size of the buffer (in bytes) for data sent over the UDP protocol.

MSS configuration example:

```
static const int mss_max = 1460;
static const int mss_min = 100;
static const char* mss_max_opt_name = "net.inet.tcp.mssdflt";
static const char* mss_min_opt_name = "net.inet.tcp.minmss";
int main(void)
{
...
    if ((sysctlbyname(mss_max_opt_name, NULL, NULL, &mss_max, sizeof(mss_max)) != 0) ||
       (sysctlbyname(mss_min_opt_name, NULL, NULL, &mss_min, sizeof(mss_min)) != 0))
    {
        ERROR(START, "Can't set tcp default maximum/minimum MSS value.");
        return EXIT_FAILURE;
    }
}
```

IPC and transport

Creating IPC channels

IPC channels can be created statically or dynamically.

Statically created IPC channels

Static creation of IPC channel has the following characteristics:

- Use of static IPC channels is not recommended for applications. Use of dynamic IPC channels is recommended (see below).
- If a KasperskyOS-based solution uses programs that for some reason cannot be packed into KPA packages, you can start these programs by using the kl.ProductRomfsLoader system program, which starts programs based on the init description and creates static IPC channels. For more details, see the examples included in the SDK.
- Creation of an IPC channel is performed by the parent process that starts the client and server (this is normally kl.ProductRomfsLoader system program).
- The client and server are not yet running when the IPC channel is created.
- You cannot create a new IPC channel in place of a deleted one.

The IPC channels defined in the init description are statically created. You can also use the task.h API to statically create IPC channels.

To get the client and server IPC handles and the endpoint ID (RIID), use the sl-static.h API.

Dynamically created IPC channels

Dynamic creation of IPC channels lets you change the topology of interaction between processes on the fly. For example, this is necessary if it is unknown which specific server provides the endpoint required by the client.

Dynamic creation of IPC channel has the following characteristics:

- The IPC channel is jointly created by the client and server.
- The client and server are already running when the IPC channel is created.
- You can create a new IPC channel in place of a deleted one.

To dynamically create IPC channels, use the DCM system program.

If a dynamically created IPC channel is no longer required, its client and server handles can be closed. The IPC channel can be created again if necessary.

Adding an endpoint from KasperskyOS Community Edition SDK to a solution

To ensure that a `Client` program can use some specific functionality via the IPC mechanism, the following is required:

1. In KasperskyOS Community Edition SDK, find the executable file (we'll call it `Server`) that implements the necessary functionality. (The term "functionality" used here refers to one or more endpoints that have their own IPC interfaces.)
2. Include the CMake package containing the `Server` file and its client library.
3. Add the `Server` executable file to the solution image.
4. In the `Client` program code, use the DCM system program to dynamically create IPC channels.
5. Edit the PSL description to allow startup of the client process, its interaction with DCM and IPC interaction between the client and the server.
6. In the source code of the `Client` program, include the server methods header file.
7. Link the `Client` program with the client library.

Creating and using your own endpoints

Overview: IPC message structure

In KasperskyOS, all interactions between processes have statically defined types. The permissible structures of an IPC message are defined by the IDL descriptions.

An IPC message (request and response) contains a constant part and an (optional) arena.

Constant part of an IPC message

The *constant part of an IPC message* contains the RIID, MID, and (optionally) fixed-size parameters of interface methods.

Fixed-size parameters are parameters that have IDL types of a fixed size.

The RIID and MID identify the interface and method being called:

- The RIID (Runtime Implementation ID) is the sequence number of the utilized endpoint within the set of server endpoints (starting at zero).
- The MID (Method ID) is the sequence number of the called method within the set of methods of the utilized endpoint (starting at zero).

The type of the fixed part of an IPC message is an element of transport code. For more details, refer to Example generation of transport methods and types.

IPC message arena

An *IPC message arena* (hereinafter also referred to as an *arena*) contains variable-size parameters of interface methods (and/or elements of these parameters).

Variable-size parameters are parameters that have IDL types of a variable size.

For more details, refer to Working with an IPC message arena.

Maximum IPC message size

The maximum size of an IPC message is determined by the KasperskyOS kernel parameters. On most hardware platforms supported by KasperskyOS, the cumulative size of the constant part and arena of an IPC message cannot exceed 4, 8, or 16 MB.

IPC message structure verification by the security module

Prior to querying IPC message-related rules, the Kaspersky Security Module verifies that the sent IPC message is correct. Requests and responses are both validated. If the IPC message has an incorrect structure, it will be rejected without calling the security model methods associated with it.

Example generation of transport methods and types

When building a solution, the transport code is generated from the formal specifications of solution components. As an example, we will examine the `Server` process class that provides the `FS` endpoint, which contains a single `Open()` method:

Server.edl

```
task class Server

/* OpsComp is the named instance of the Operations component */
components {
    OpsComp: Operations
}
```

Operations.cdl

```
component Operations

/* FS is the local name of the endpoint implementing the Filesystem interface */
endpoints {
    FS: Filesystem
}
```

Filesystem.idl

```
package Filesystem

interface {
    Open(in string<256> name, out UInt32 h);
}
```

These descriptions will be used to generate the files named `Server.edl.h`, `Operations.cdl.h`, and `Filesystem.idl.h`, which contain the following methods and types:

Methods and types that are common to the client and server

- **Abstract interfaces containing the pointers to the implementations of the methods included in them.**

  In our example, one abstract interface (`Filesystem`) will be generated:

```
typedef struct Filesystem {
            const struct Filesystem_ops *ops;
        } Filesystem;

typedef nk_err_t
Filesystem_Open_fn(struct Filesystem *, const
                   struct Filesystem_Open_req *,
                   const struct nk_arena *,
                   struct Filesystem_Open_res *,
                   struct nk_arena *);

typedef struct Filesystem_ops {
            Filesystem_Open_fn *Open;
        } Filesystem_ops;
```

- **Set of interface methods.**

  When calling an interface method, the corresponding values of the RIID and MID are automatically inserted into the request.

  In our example, a single `Filesystem_Open` interface method will be generated:

```
nk_err_t Filesystem_Open(struct Filesystem *self,
                         struct Filesystem_Open_req *req,
                         const
                         struct nk_arena *req_arena,
                         struct Filesystem_Open_res *res,
                         struct nk_arena *res_arena)
```

Methods and types used only on the client

- **Types of proxy objects.**

  A proxy object is used as an argument in an interface method. In our example, a single `Filesystem_proxy` proxy object type will be generated:

```
typedef struct Filesystem_proxy {
            struct Filesystem base;
            struct nk_transport *transport;
            nk_iid_t iid;
        } Filesystem_proxy;
```

- **Functions for initializing proxy objects.**

  In our example, the single initializing function `Filesystem_proxy_init` will be generated:

```
void Filesystem_proxy_init(struct Filesystem_proxy *self,
                           struct nk_transport *transport,
                           nk_iid_t iid)
```

- **Types that define the structure of the constant part of a message for each specific method.**

  In our example, two such types will be generated: `Filesystem_Open_req` (for a request) and `Filesystem_Open_res` (for a response).

```
typedef struct __nk_packed Filesystem_Open_req {
            __nk_alignas(8)
            struct nk_message base_;
            __nk_alignas(4) nk_ptr_t name;
        } Filesystem_Open_req;

typedef struct Filesystem_Open_res {
            union {
                struct {
                    __nk_alignas(8)
                    struct nk_message base_;
                    __nk_alignas(4) nk_uint32_t h;
                };
                struct {
                    __nk_alignas(8)
                    struct nk_message base_;
                    __nk_alignas(4) nk_uint32_t h;
                } res_;
                struct Filesystem_Open_err err_;
            };
        } Filesystem_Open_res;
```

Methods and types used only on the server

- **Type containing all endpoints of a component, and the initializing function. (For each server component.)**

  > If there are embedded components, this type also contains their instances, and the initializing function takes their corresponding initialized structures. Therefore, if embedded components are present, their initialization must begin with the most deeply embedded component.

In our example, the `Operations_component` structure and `Operations_component_init` function will be generated:

```
typedef struct Operations_component {
     struct Filesystem *FS;
};

void Operations_component_init(struct Operations_component *self,
                               struct Filesystem *FS)
```

- **Type containing all endpoints provided directly by the server; all instances of components included in the server; and the initializing function.**

  In our example, the `Server_entity` structure and `Server_entity_init` function will be generated:

```
#define Server_entity Server_component

typedef struct Server_component {
            struct : Operations_component *OpsComp;
        } Server_component;

void Server_entity_init(struct Server_entity *self,
                            struct Operations_component *OpsComp)
```

- **Types that define the structure of the constant part of a message for any method of a specific interface.**

  In our example, two such types will be generated: `Filesystem_req` (for a request) and `Filesystem_res` (for a response).

```
typedef union Filesystem_req {
    struct nk_message base_;
    struct Filesystem_Open_req Open;
};

typedef union Filesystem_res {
    struct nk_message base_;
    struct Filesystem_Open_res Open;
};
```

- **Types that define the structure of the constant part of a message for any method of any endpoint of a specific component.**

  > If embedded components are present, these types also contain structures of the constant part of a message for any method of any endpoint included in all embedded components.

  In our example, two such types will be generated: `Operations_component_req` (for a request) and `Operations_component_res` (for a response).

```
typedef union Operations_component_req {
            struct nk_message base_;
            Filesystem_req FS;
        } Operations_component_req;

typedef union Operations_component_res {
            struct nk_message base_;
            Filesystem_res FS;
        } Operations_component_res;
```

- **Types that define the structure of the constant part of a message for any method of any endpoint of a specific component whose instance is included in the server.**

> If embedded components are present, these types also contain structures of the constant part of a message for any method of any endpoint included in all embedded components.

In our example, two such types will be generated: `Server_entity_req` (for a request) and `Server_entity_res` (for a response).

```
#define Server_entity_req Server_component_req

typedef union Server_component_req {
            struct nk_message base_;
            Filesystem_req OpsComp_FS;
        } Server_component_req;

#define Server_entity_res Server_component_res

typedef union Server_component_res {
            struct nk_message base_;
            Filesystem_res OpsComp_FS;
        } Server_component_res;
```

- **Dispatch methods (dispatchers) for a separate interface, component, or process class.**

  Dispatchers analyze the received query (the RIID and MID values), call the implementation of the corresponding method, and then save the response in the buffer. In our example, three dispatchers will be generated: `Filesystem_interface_dispatch`, `Operations_component_dispatch`, and `Server_entity_dispatch`.

  The process class dispatcher handles the request and calls the methods implemented by this class. If the request contains an incorrect RIID (for example, an RIID for a different endpoint that this process class does not have) or an incorrect MID, the dispatcher returns `NK_EOK` or `NK_ENOENT`.

```
nk_err_t Server_entity_dispatch(struct Server_entity *self,
                                const
                                struct nk_message *req,
                                const
                                struct nk_arena *req_arena,
                                struct nk_message *res,
                                struct nk_arena *res_arena)
```

  In special cases, you can use dispatchers of the interface and the component. They take an additional argument: interface implementation ID (`nk_iid_t`). The request will be handled only if the passed argument and RIID from the request match, and if the MID is correct. Otherwise, the dispatchers return `NK_EOK` or `NK_ENOENT`.

```
nk_err_t Operations_component_dispatch(struct Operations_component *self,
                                       nk_iid_t iidOffset,
                                       const
                                       struct nk_message *req,
                                       const
                                       struct nk_arena *req_arena,
                                       struct nk_message *res,
                                       struct nk_arena *res_arena)

nk_err_t Filesystem_interface_dispatch(struct Filesystem *impl,
                                       nk_iid_t iid,
                                       const
                                       struct nk_message *req,
                                       const
                                       struct nk_arena *req_arena,
                                       struct nk_message *res,
                                       struct nk_arena *res_arena)
```

Working with an IPC message arena

Arena overview

From the perspective of a developer of KasperskyOS-based solutions, an IPC message arena is a byte buffer in the memory of a process intended for storing variable-size data transmitted over IPC. This variable-size data includes input parameters, output parameters, and error parameters of interface methods (and/or elements of these parameters) that have variable-size IDL types. An arena is also used when querying the Kaspersky Security Module to store input parameters of security interface methods (and/or elements of these parameters) that have variable-size IDL types. (Parameters of fixed-size interface methods are stored in the constant part of an IPC message.) Arenas are used on the client side and on the server side. One arena is intended either for transmitting or for receiving variable-size data through IPC, but not for both transmitting and receiving this data at the same time. In other words, arenas can be divided into IPC request arenas (containing input parameters of interface methods) and IPC response arenas (containing output parameters and error parameters of interface methods).

Only the utilized part of an arena that is occupied by data is transmitted over IPC. (If it has no data, the arena is not transmitted.) The utilized part of an arena includes one or more segments. One segment of an arena stores an array of same-type objects, such as an array of single-byte objects or an array of structures. Arrays of different types of objects may be stored in different segments of an arena. The starting address of an arena must be equal to the boundary of a *2^N*-byte sequence, where *2^N* is a value that is greater than or equal to the size of the largest primitive type in the arena (for example, the largest field of a primitive type in a structure). The address of an arena chunk must also be equal to the boundary of a *2^N*-byte sequence, where *2^N* is a value that is greater than or equal to the size of the largest primitive type in the arena chunk.

An arena must have multiple segments if the interface method has multiple variable-size input, output, or error parameters, or if multiple elements of input, output, or error parameters of the interface method have a variable size. For example, if an interface method has an input parameter of the `sequence` IDL type and an input parameter of the `bytes` IDL type, the IPC request arena will have at least two segments. In this case, it may even require additional segments if a parameter of the `sequence` IDL type consists of elements of a variable-size IDL type (for example, if elements of a sequence are `string` buffers). As another example, if an interface method has one output parameter of the `struct` IDL type that contains two fields of the `bytes` and `string` type, the IPC response arena will have two segments.

Due to the alignment of arena chunk addresses, there may be unused intervals between these chunks. Therefore, the size of the utilized part of an arena may exceed the size of the data it contains.

API for working with an arena

The set of functions and macros for working with an arena is defined in the header file `sysroot-*-kos/include/nk/arena.h` from the KasperskyOS SDK. The `NkKosCopyStringToArena()` function from the transport-kos.h API is used to copy a string to the arena. Additionally, to work with strings transferred in the arena, you can use the API that is defined in the header file `sysroot-*-kos/include/services/rtl/nk_cstr.h` from the KasperskyOS SDK.

In the API functions and macros, an arena and arena chunk are identified by an arena descriptor (the `nk_arena` type) and an arena chunk descriptor (the `nk_ptr_t` type), respectively. An *arena descriptor* is a structure containing three pointers: one pointer to the start of the arena, one pointer to the start of the unused part of the arena, and one pointer to the end of the arena. An *arena chunk descriptor* is a structure containing the offset of the arena chunk in bytes (relative to the start of the arena) and the size of the arena chunk in bytes. (The arena chunk descriptor type is defined in the header file `sysroot-*-kos/include/nk/types.h` from the KasperskyOS SDK.)

> You should not use functions from the header file `sysroot-*-kos/include/nk/arena.h` whose names begin with the `_` or `__` character because these functions are internal implementation details that may change.
> On hardware platforms running an Arm processor architecture, input and output parameters of the API cannot be saved in "Device memory" because this could lead to unpredictable behavior. API parameters must be saved in "Normal memory". To copy data from "Device memory" to "Normal memory" and vice versa, you must use the `RtlPedanticMemcpy()` function declared in the header file `sysroot-*-kos/include/rtl/string_pedantic.h` from the KasperskyOS SDK.

Information on the functions and macros defined in the header file `sysroot-*-kos/include/nk/arena.h` is provided in the table below.

Creating an arena

To pass variable-size parameters of interface methods over IPC, you must create arenas on the client side and on the server side. (When IPC requests are handled on the server side using the `NkKosDoDispatch()` or `NkKosDoDispatchEx()` functions declared in the header file `sysroot-*-kos/include/coresrv/nk/transport-kos-dispatch.h` from the KasperskyOS SDK, the IPC request arena and IPC response arena are created automatically.)

To create an arena, you must create a buffer (in the stack or heap) and initialize the arena descriptor.

The address of the buffer must be aligned to comply with the maximum size of a primitive type that can be put into this buffer. The address of a dynamically created buffer usually has adequate alignment to hold the maximum amount of data of the primitive type. To ensure the required alignment of the address of a statically created buffer, you can use the `alignas` specifier.

To initialize an arena descriptor using the pointer to an already created buffer, you must use an API function or macro:

- `NK_ARENA_INITIALIZER()` macro
- `nk_arena_init()` function
- `nk_arena_create()` function
- `NK_ARENA_FINAL()` macro
- `nk_arena_init_final()` macro

The type of pointer makes no difference because this pointer is converted into a pointer to a single-byte object in the code of API functions and macros.

The `NK_ARENA_INITIALIZER()` macro and the `nk_arena_init()` and `nk_arena_create()` functions initialize such arena descriptor that may contain one or more segments. The `NK_ARENA_FINAL()` and `nk_arena_init_final()` macros initialize the descriptor of this arena, which contains only one chunk spanning the entire arena throughout its entire lifetime.

To create a buffer in the stack and initialize the arena descriptor in one step, use the `NK_ARENA_AUTO()` macro. This macro creates an arena that may contain one or more segments, and the address of the buffer created by this macro has adequate alignment to hold the maximum amount of data of the primitive type.

The size of an arena must be sufficient to hold variable-size parameters for IPC requests or IPC responses of one interface method or a set of interface methods corresponding to one interface, component, or process class while accounting for the alignment of segment addresses. Automatically generated transport code (the header files `*.idl.h`, `*.cdl.h`, and `*.edl.h`) contain `*_arena_size` constants whose values are guaranteed to comply with sufficient sizes of arenas in bytes.

The header files `*.idl.h`, `*.cdl.h`, and `*.edl.h` contain the following `*_arena_size` constants:

- `<interface name>_<interface method name>_req_arena_size` – size of an IPC request arena for the specified interface method of the specified interface
- `<interface name>_<interface method name>_res_arena_size` – size of an IPC response arena for the specified interface method of the specified interface
- `<interface name>_req_arena_size` – size of an IPC request arena for any interface method of the specified interface
- `<interface name>_res_arena_size` – size of an IPC response arena for any interface method of the specified interface

The header files `*.cdl.h` and `*.edl.h` also contain the following `*_arena_size` constants:

- `<component name>_component_req_arena_size` – size of an IPC request arena for any interface method of the specified component
- `<component name>_component_res_arena_size` – size of an IPC response arena for any interface method of the specified component

The `*.edl.h` header files also contain the following `*_arena_size` constants:

- `<process class name>_entity_req_arena_size` – size of an IPC request arena for any interface method of the specified process class
- `<process class name>_entity_res_arena_size` – size of an IPC response arena for any interface method of the specified process class

Constants containing the size of an IPC request arena or IPC response arena for one interface method (`<interface name>_<interface method name>_req_arena_size` and `<interface name>_<interface method name>_res_arena_size`) are intended for use on the client side. All other constants can be used on the client side and on the server side.

Examples of creating an arena:

```
/* Example 1 */
alignas(8) char reqBuffer[Write_WriteInLog_req_arena_size];
struct nk_arena reqArena = NK_ARENA_INITIALIZER(
                           reqBuffer, reqBuffer + sizeof(reqBuffer));
```

```
/* Example 2 */
struct nk_arena res_arena;
char res_buf[kl_rump_DhcpcdConfig_GetOptionNtpServers_res_arena_size];
nk_arena_init(&res_arena, res_buf, res_buf + sizeof(res_buf));

/* Example 3 */
char req_buffer[kl_CliApplication_Run_req_arena_size];
struct nk_arena req_arena = nk_arena_create(req_buffer, sizeof(req_buffer));

/* Example 4 */
nk_ptr_t ptr;
const char *cstr = "example";
nk_arena arena = NK_ARENA_FINAL(&ptr, cstr, strlen(cstr) + 1);

/* Example 5 */
const char *path = "path_to_file";
size_t len = strlen(path);
/* Structure for saving the constant part of an IPC request */
struct kl_VfsFilesystem_Rmdir_req req;
struct nk_arena req_arena;
nk_arena_init_final(&req_arena, &req.path, path, len + 1);

/* Example 6 */
struct nk_arena res_arena = NK_ARENA_AUTO(kl_drivers_Example_GetInfo_res_arena_size);
```

Adding data to an arena before transmission over IPC

Before transmitting an IPC request on the client side or an IPC response on the server side, data must be added to the arena. If the `NK_ARENA_FINAL()` macro or the `nk_arena_init_final()` macro is used to create an arena, you do not need to reserve an arena chunk. Instead, you only need to add data to this chunk. If the `NK_ARENA_INITIALIZER()` or `NK_ARENA_AUTO()` macro, or the `nk_arena_init()` or `nk_arena_create()` function is used to create an arena, one or multiple segments in the arena must be reserved to hold data. To reserve an arena chunk, you must use an API function or macro:

- `nk_arena_alloc_aligned()` macro
- `nk_arena_alloc()` macro
- `nk_arena_store_aligned()` macro
- `nk_arena_store()` macro
- `NkKosCopyStringToArena()` function

The arena chunk descriptor, which is passed through the output parameter of these macros and functions and through the output parameter of the `NK_ARENA_FINAL()` and `nk_arena_init_final()` macros, must be put into the constant part or into the arena of an IPC message. If an interface method has a variable-size parameter, the constant part of IPC messages contains arena chunk descriptor containing the parameter instead of the actual parameter. If an interface method has a fixed-size parameter with variable-size elements, the constant part of IPC messages contains arena chunk descriptors containing the parameter elements instead of the actual parameter elements. If an interface method has a variable-size parameter containing variable-size elements, the constant part of IPC messages contains arena chunk descriptor containing the descriptors of other arena chunks that contain these parameter elements.

The `nk_arena_store_aligned()` and `nk_arena_store()` macros and the `NkKosCopyStringToArena()` function not only reserve an arena chunk, but also copy data to this chunk.

The `nk_arena_alloc_aligned()` and `nk_arena_alloc()` macros get the address of a reserved arena chunk. You can also get an arena chunk address by using the `nk_arena_get()` macro, which additionally uses the output parameter to pass the quantity of objects accommodated in this chunk.

A reserved arena chunk can be reduced. To do so, you need to use the `nk_arena_shrink()` macro.

To undo a current reservation of arena chunks so that new chunks can be reserved for other data (after sending an IPC message), call the `nk_arena_reset()` function. If the `NK_ARENA_FINAL()` macro or `nk_arena_init_final()` macro is used to create an arena, you do not need to undo a segment reservation because the arena contains one segment spanning the entire arena throughout its entire life cycle.

Examples of adding data to an arena:

```
/* Example 1 */
char req_buffer[kl_rump_NpfctlFilter_TableAdd_req_arena_size];
struct nk_arena req_arena = NK_ARENA_INITIALIZER(req_buffer, req_buffer + sizeof(req_buffer));
/* Structure for saving the constant part of an IPC request */
struct kl_rump_NpfctlFilter_TableAdd_req req;
if (nk_arena_store(char, &req_arena, &req.tid, tid, tidlen))
    return ENOMEM;
if (nk_arena_store(char, &req_arena, &req.cidrAddr, cidr_addr, cidr_addrlen))
    return ENOMEM;

/* Example 2 */
char req_arena_buf[StringMaxSize];
struct nk_arena req_arena = NK_ARENA_INITIALIZER(req_arena_buf,
                                        req_arena_buf + sizeof(req_arena_buf));
/* Structure for saving the constant part of an IPC request */
kl_drivers_FBConsole_SetFont_req req;
size_t buf_size = strlen(fileName) + 1;
char *buf = nk_arena_alloc(char, &req_arena, &req.fileName, buf_size);
memcpy(buf, fileName, buf_size);

/* Example 3 */
char reqArenaBuf[kl_core_DCM_req_arena_size];
struct nk_arena reqArena
    = NK_ARENA_INITIALIZER(reqArenaBuf,
                           reqArenaBuf + sizeof(reqArenaBuf));
/* Structure for saving the constant part of an IPC request */
kl_core_DCM_Subscribe_req req;
rc = NkKosCopyStringToArena(&reqArena, &req.endpointType, endpointType);
if (rc != rcOk)
    return rc;
rc = NkKosCopyStringToArena(&reqArena, &req.endpointName, endpointName);
if (rc != rcOk)
    return rc;
rc = NkKosCopyStringToArena(&reqArena, &req.serverName, serverName);
if (rc != rcOk)
    return rc;

/* Example 4 */
unsigned counter = 0;
nk_ptr_t *paths;
/* Reserve an arena chunk for descriptors of other arena chunks */
paths = nk_arena_alloc(nk_ptr_t, resArena, &res->logRes, msgCount);
while(...)
{
    ...
    /* Reserve arena chunks and save their descriptors in
     * a previously reserved arena chunk with the paths address */
    char *str = nk_arena_alloc(
                                        char,
                                        resArena,
                                        &paths[counter],
                                        stringLength + 1);
    if (str == NK_NULL)
        return NK_ENOMEM;
    snprintf(str, (stringLength + 1), "%s", buffer);
    ...
    counter++;
}
```

Retrieving data from an arena after receipt over IPC

Prior to receiving an IPC request on the server side or an IPC response on the client side for an arena that will store the data received over IPC, you must undo the current reservation of segments by calling the `nk_arena_reset()` function. This must be done even if the `NK_ARENA_FINAL()` macro or the `nk_arena_init_final()` macro is used to create the arena. (The `NK_ARENA_INITIALIZER()` and `NK_ARENA_AUTO()` macros, and the `nk_arena_init()` and `nk_arena_create()` functions create an arena without reserved segments. You do not need to call the `nk_arena_reset()` function if this arena will only be used once to save data received over IPC.)

You do not need to free the currently reserved arena chunks if one of the syscalls.h API functions is directly used to receive an IPC message. However, you must do that prior to calling transport code methods on the client side and the `nk_transport_recv()` function on the server side. (The `nk_transport_recv()` function is declared in the header file `sysroot-*-kos/include/nk/transport.h` from the KasperskyOS SDK.)

When calling the `NkKosTransport_Dispatch()` function declared in the header file `sysroot-*-kos/include/coresrv/nk/transport-kos.h` from the KasperskyOS SDK, you must specify arenas that do not have reserved chunks. If the `NK_ARENA_FINAL()` or `nk_arena_init_final()` macro is used to create these arenas, call the `nk_arena_reset()` function to free the reserved chunks.

To get pointers to arena chunks and the quantity of objects accommodated in these chunks, you must use the `nk_arena_get()` macro while using the input parameter to pass the corresponding arena chunk descriptors received from the constant part and arena of the IPC message.

Example of receiving data from an arena:

```
struct nk_arena res_arena;
char res_buf[kl_rump_DhcpcdConfig_Version_res_ver_size];
nk_arena_init(&res_arena, res_buf, res_buf + sizeof(res_buf));
/* Structure for saving an IPC request */
struct kl_rump_DhcpcdConfig_Version_req req;
req.buflen = buflen;
/* Structure for saving an IPC response */
struct kl_rump_DhcpcdConfig_Version_res res;
/* Call the interface method */
if (kl_rump_DhcpcdConfig_Version(dhcpcd.proxy, &req, NULL, &res, &res_arena) != NK_EOK)
    return -1;
size_t ptrlen;
char *ptr = nk_arena_get(char, &res_arena, &res.ver, &ptrlen);
memcpy(buf, ptr, ptrlen);
```

Additional capabilities of the API

To get the arena size, call the `nk_arena_capacity()` function.

To get the size of the utilized part of the arena, call the `nk_arena_allocated_size()` function.

To verify that the arena chunk descriptor is correct, use the `nk_arena_validate()` macro.

Information about API functions and macros

Functions and macros of arena.h

Function/Macro	Information about the function/macro
`NK_ARENA_INITIALIZER()`	Purpose Initializes the arena descriptor. Parameters • [in] `_start` – pointer to the start of the arena. • [in] `_end` – pointer to the end of the arena. Macro values Code that initializes the arena descriptor.
`nk_arena_init()`	Purpose Initializes the arena descriptor. Parameters • [out] `self` – pointer to the arena descriptor. • [in] `start` – pointer to the start of the arena. • [in] `end` – pointer to the end of the arena. Returned values N/A
`nk_arena_create()`	Purpose Creates and initializes the arena descriptor. Parameters • [in] `start` – pointer to the start of the arena. • [in] `size` – arena size (in bytes). Returned values Arena descriptor.
`NK_ARENA_AUTO()`	Purpose Creates a buffer in the stack, and creates and initializes the arena descriptor. Parameters • [in] `size` – arena size (in bytes). It must be defined as a constant. Macro values Arena descriptor.

Function/Macro	Information about the function/macro
`NK_ARENA_FINAL()`	Purpose Initializes the arena descriptor containing only one segment. Parameters • [out] `ptr` – pointer to the arena chunk descriptor. • [in] `start` – pointer to the start of the arena. • [in] `count` – number of objects in the arena chunk. Macro values Arena descriptor.
`nk_arena_reset()`	Purpose Resets the reservation of arena chunks. Parameters • [in,out] `self` – pointer to the arena descriptor. Returned values N/A
`nk_arena_capacity()`	Purpose Gets the size of an arena. Parameters • [in] `self` – pointer to the arena descriptor. Returned values Size of the arena, in bytes. Additional information If the parameter has the `NK_NULL` value, it returns `0`.

Function/Macro	Information about the function/macro
`nk_arena_validate()`	Purpose Verifies that the arena chunk descriptor is correct. Parameters • [in] `type` – type of objects in the arena chunk. • [in] `arena` – pointer to the arena descriptor. • [in] `ptr` – pointer to the arena chunk descriptor. Macro values It has a value of `0` when the arena size is not zero if all of the following conditions are fulfilled: 1. The offset specified in the arena chunk descriptor does not exceed the arena size. 2. The size specified in the arena chunk descriptor does not exceed the arena size reduced by the offset specified in the arena chunk descriptor. 3. The size specified in the arena chunk descriptor is a multiple of the size of the type of objects in this arena chunk. It has a value of `0` when the arena size is zero if all of the following conditions are fulfilled: 1. The offset specified in the arena chunk descriptor is equal to zero. 2. The size specified in the arena chunk descriptor is equal to zero. It has a value of `-1` if even one of the conditions is not fulfilled (regardless of whether the arena size is zero or non-zero), or if the `ptr` parameter has the `NK_NULL` value.
`nk_arena_allocated_size()`	Purpose Gets the size of the utilized part of the arena. Parameters • [in] `self` – pointer to the arena descriptor. Returned values Size of the utilized part of the arena, in bytes. Additional information If the parameter has the `NK_NULL` value, it returns `0`.
`nk_arena_init_final()`	Purpose Initializes the arena descriptor containing only one segment. Parameters • [out] `arena` – pointer to the arena descriptor. • [out] `ptr` – pointer to the arena chunk descriptor. • [in] `start` – pointer to the start of the arena. • [in] `count` – number of objects in the arena chunk. Macro values N/A

Function/Macro	Information about the function/macro
`nk_arena_alloc_aligned()`	Purpose Reserves an arena chunk with the specified alignment for the defined number of objects of the specified type. Parameters • [in] `type` – type of objects in the arena chunk. • [in,out] `arena` – pointer to the arena descriptor. • [out] `ptr` – pointer to the arena chunk descriptor. • [in] `count` – number of objects in the arena chunk. • [in] `align` – value defining the arena chunk alignment. The arena chunk address can be unaligned (*align=1*) or aligned (*align=2,4,...,2^N*) to the boundary of a *2^N*-byte sequence (for example, two-byte or four-byte). Macro values It has the address of the reserved arena chunk if successful, otherwise `NK_NULL`.
`nk_arena_alloc()`	Purpose Reserves an arena chunk for the defined number of objects of the specified type. Parameters • [in] `type` – type of objects in the arena chunk. • [in,out] `arena` – pointer to the arena descriptor. • [out] `ptr` – pointer to the arena chunk descriptor. • [in] `count` – number of objects in the arena chunk. Macro values It has the address of the reserved arena chunk if successful, otherwise `NK_NULL`.
`nk_arena_store_aligned()`	Purpose Reserves an arena chunk with the specified alignment for the defined number of objects of the specified type and copies these objects to the reserved chunk. Parameters • [in] `type` – type of objects in the arena chunk. • [in,out] `arena` – pointer to the arena descriptor. • [out] `ptr` – pointer to the arena chunk descriptor. • [in] `src` – pointer to the buffer containing the objects that need to be copied to the arena chunk. • [in] `count` – number of objects in the arena chunk. • [in] `align` – value defining the arena chunk alignment. The arena chunk address can be unaligned (*align=1*) or aligned (*align=2,4,...,2^N*) to the boundary of a *2^N*-byte sequence (for example, two-byte or four-byte). Macro values It has a value of `0` if successful, otherwise it has a value of `-1`.

Function/Macro	Information about the function/macro
`nk_arena_store()`	Purpose Reserves an arena chunk for the specified number of objects of the defined type and copies these objects to the reserved chunk. Parameters • [in] `type` – type of objects in the arena chunk. • [in,out] `arena` – pointer to the arena descriptor. • [out] `ptr` – pointer to the arena chunk descriptor. • [in] `src` – pointer to the buffer containing the objects that need to be copied to the arena chunk. • [in] `count` – number of objects in the arena chunk. Macro values It has a value of `0` if successful, otherwise it has a value of `-1`.
`nk_arena_get()`	Purpose Gets the address of an arena chunk and the quantity of objects of the specified type in this chunk. Parameters • [in] `type` – type of objects in the arena chunk. • [in] `arena` – pointer to the arena descriptor. • [in] `ptr` – pointer to the arena chunk descriptor. • [out] `count` – pointer to the number of objects in the arena chunk. Macro values It has the address of the arena chunk if successful, otherwise `NK_NULL`. Additional information If the size of the arena chunk is not a multiple of the size of the type of objects in this chunk, it has the `NK_NULL` value.
`nk_arena_shrink()`	Purpose Reduces the size of an arena chunk. Parameters • [in] `type` – type of objects in the arena chunk. • [in,out] `arena` – pointer to the arena descriptor. • [in,out] `ptr` – pointer to the arena chunk descriptor. • [in] `count` – quantity of objects in the reduced arena chunk. Macro values It has the address of the reduced arena chunk if successful, otherwise `NK_NULL`. Additional information If the required size of the arena chunk exceeds the current size, it has the `NK_NULL` value. If the alignment of the arena chunk that needs to be reduced does not match the type of objects for which the reduced chunk is intended, it has the `NK_NULL` value. If the arena chunk that needs to be reduced is the last chunk in the arena, the freed part of this chunk will become available for reservation of subsequent chunks.

Transport code in C++

> Before reading this section, you should review the information on the IPC mechanism and formal specifications of the solution components.

Implementation of interprocess interaction requires transport code, which is responsible for generating, sending, receiving, and processing IPC messages.

Transport code

The transport code includes proxies and stubs for use by both the client and the server. Proxy objects are used by the client to pack the parameters of the called method into an IPC request, execute the IPC request, and unpack the IPC response. Stubs are used by the server to unpack the parameters from the IPC request, dispatch the call to the appropriate method implementation, and pack the IPC response.

Generating transport code for development in C++

The solution developer does not need to manually write the transport code. A transport code generator is provided as part of the KasperskyOS Community Edition SDK. This generator creates transport code from the formal specifications of solution components.

To generate transport code in C++ when building a solution, you need to use the `CMake` commands add_nk_idl(), add_nk_cdl(), and add_nk_edl().

C++ types in the *.idl.cpp.h file

Each interface is defined in an IDL description. This description defines the interface name, signatures of interface methods, and data types for the parameters of interface methods.

The `CMake` command add_nk_idl() creates a `CMake` target to generate header files for the specified IDL file. The generated header files contain a C++ representation for the interface and data types described in the IDL file, and the methods required for use of proxy objects and stubs.

The mapping of data types declared in an IDL file to C++ types is presented in the table below.

Mapping IDL types to C++ types

IDL type	C++ type
`SInt8`	`int8_t`
`SInt16`	`int16_t`
`SInt32`	`int32_t`
`SInt64`	`int64_t`

IDL type	C++ type
UInt8	uint8_t
UInt16	uint16_t
UInt32	uint32_t
UInt64	uint64_t
Handle	Handle (defined in sysroot-*-kos/include/handle/handletype.h)
string	std::string
union	std::variant
struct	struct
array	std::array
sequence	std::vector
bytes	std::vector<std::byte>

Working with transport code in C++

The scenarios for developing a client and server that exchange IPC messages are presented in the section titled "Dynamically creating IPC channels for C++ development".

Dynamically creating IPC channels for C++ development using DCM

The system program named `DCM` included in the KasperskyOS SDK is used to dynamically create IPC channels for C++ development. The API of this program is provided in the section titled Dynamically creating IPC channels using the DCM system program.

Dynamic creation of an IPC channel on the client side includes the following steps:

1. Include the generated description header file (`*.edl.cpp.h`) in the client program.
2. Include the generated header files of the descriptions of the utilized interfaces (`*.idl.cpp.h`).
3. Include the header files from the KasperskyOS SDK:
   - `/opt/KasperskyOS-Community-Edition-<platform>-<version>/sysroot-*-kos/include/kosipc/application.h`
   - `/opt/KasperskyOS-Community-Edition-<platform>-<version>/sysroot-*-kos/include/kosipc/make_application.h`
   - `/opt/KasperskyOS-Community-Edition-<platform>-<version>/sysroot-*-kos/include/kosipc/unique_ptr.h`
   - `/opt/KasperskyOS-Community-Edition-<platform>-<version>/sysroot-*-kos/include/kosipc/connect_dcm_publication.h`
4. Create an application object by calling the `kosipc::MakeApplicationAutodetect()` function. (You can also use the `kosipc::MakeApplication()` and `kosipc::MakeApplicationPureClient()` functions.)

5. Create and initialize the proxy object for the endpoint that implements the required interface:

   - Call the `MakeProxy<Interface>()` function.
   - Use the `kosipc::ConnectDcmPublication()` function call as the input parameter of the `MakeProxy<Interface>()` function.
   - Pass the following parameters to the `kosipc::ConnectDcmPublication()` function:

     a. [Optional] Server name, or `std::nullopt` so that the received notifications about endpoint publication on the server are not filtered by server name.

     b. [Optional] Qualified name of the endpoint, or `std::nullopt` so that the received notifications about endpoint publication on the server are not filtered based on the qualified name of the endpoint.

     c. [Optional] Handle identifying the server, or `std::nullopt` so that the received notifications about endpoint publication on the server are not filtered based on a specific server.

     d. [Optional] Endpoint search timeout, or `std::nullopt` to define an unlimited timeout. Default value: `std::nullopt`.

     e. [Optional] Timeout for the server response to the connection request, or `std::nullopt` to define an unlimited timeout. Default value: `std::nullopt`.

   If none of the parameters are specified, the `kosipc::ConnectDcmPublication()` function will attempt to establish a connection with the server's first available endpoint that implements the required interface.

   If the server-identifying handle and qualified name of the endpoint are defined during the function call, the request to create an IPC channel with the server to use the defined endpoint with the required interface will be sent immediately without searching for notifications about publishing or depublishing this endpoint.

After successful initialization of the proxy object, the client can call endpoint methods.

Example

```
// Create an application object
kosipc::Application app = kosipc::MakeApplicationAutodetect();

// Create a proxy object for the endpoint that implements the Writer interface.
// The kosipc::ConnectDcmPublication() function will establish a connection with the first
// endpoint implementing the required interface available on the server.
// The endpoint search timeout is limited to 10 minutes.
auto writer = app.MakeProxy<Writer>(kosipc::ConnectDcmPublication(10min));

// Call the endpoint method
writer->DoSomeWork();
```

Dynamic creation of an IPC channel on the server side includes the following steps:

1. Include the generated header file (`*.edl.cpp.h`) containing a description of the component structure of the server, including all provided endpoints, in the server program.

2. Include the header files from the KasperskyOS SDK:

   - `/opt/KasperskyOS-Community-Edition-<platform>-<version>/sysroot-*-kos/include/kosipc/application.h`
   - `/opt/KasperskyOS-Community-Edition-<platform>-<version>/sysroot-*-kos/include/kosipc/event_loop.h`

- `/opt/KasperskyOS-Community-Edition-<platform>-<version>/sysroot-*-kos/include/kosipc/make_application.h`
- `/opt/KasperskyOS-Community-Edition-<platform>-<version>/sysroot-*-kos/include/kosipc/root_component.h`
- `/opt/KasperskyOS-Community-Edition-<platform>-<version>/sysroot-*-kos/include/kosipc/serve_connection_requests.h`
- `/opt/KasperskyOS-Community-Edition-<platform>-<version>/sysroot-*-kos/include/kosipc/serve_dynamic_channel.h`
- `/opt/KasperskyOS-Community-Edition-<platform>-<version>/sysroot-*-kos/include/kosipc/service_list.h`
- `/opt/KasperskyOS-Community-Edition-<platform>-<version>/sysroot-*-kos/include/kosipc/simple_connection_acceptor.h`
- `/opt/KasperskyOS-Community-Edition-<platform>-<version>/sysroot-*-kos/include/kosipc/dcm_service_publisher.h`

3. Create classes containing the implementations of interfaces that the server provides as endpoints. Create and initialize the objects of these classes.

4. Create an application object by calling the `kosipc::MakeApplicationAutodetect()` function.

5. Create and initialize the `kosipc::components::Root` class object that describes the structure of components and endpoints of the server. This structure is generated from the descriptions in the CDL and EDL files.

6. Bind the `kosipc::components::Root` class object to the class objects created at step 3.

7. Create and initialize the `kosipc::EventLoop` class object that implements a loop for dispatching incoming IPC messages by calling the `MakeEventLoop()` function. Use the `ServeDynamicChannel()` function call as an input parameter of the `MakeEventLoop()` function. Pass the `kosipc::components::Root` class object created at step 5 to the `ServeDynamicChannel()` function.

8. [Optional] Create and initialize an object of the `kosipc::ServiceList` class that contains a list of endpoints to be published on the server. To add endpoints to the list, use the `AddServices()` method. For example, use of an object of the `kosipc::ServiceList` class enables you to separate components and endpoints of the server into groups or publish endpoints under different names (see below for a code example).

9. Create and initialize an object of the `kosipc::DcmServicePublisher` class that is intended for publishing endpoints on the server. Pass the following input parameters to the constructor:

   - Application object from step 4.
   - Object of the `kosipc::components::Root` class created at step 5, or the `kosipc::ServiceList` class object generated at step 8.

> The objects created at steps 9-11 must be created after the loop for dispatching incoming IPC messages is created (see step 7).

The `kosipc` library guarantees the following: when an object of the `kosipc::DcmServicePublisher` class is created, all endpoints passed to it are published to DCM, and they are unpublished when the object is destroyed.

10. Create and initialize the object that implements the handler for receiving incoming requests to dynamically create an IPC channel.

    When creating the object, you can use the `kosipc::SimpleConnectionAcceptor` class, which is a standard implementation of the `kosipc::IConnectionAcceptor` interface. (The `kosipc::IConnectionAcceptor` interface is defined in `sysroot-*-kos/include/kosipc/connection_acceptor.h`.) In this case, the handler will implement the following logic: if the endpoint requested by the client is published on the server, the request from the client is accepted. Otherwise, it is rejected.

    If you need to create your own handler, you should implement your own request handling logic in the `OnConnectionRequest()` method inherited from the `kosipc::IConnectionAcceptor` interface. This method will be called by the server when it receives a request for dynamic IPC channel creation from the client.

11. Create a `kosipc::EventLoop` class object that implements a loop for receiving incoming requests to dynamically create an IPC channel by calling the `MakeEventLoop()` function. Use the `ServeConnectionRequests()` function call as an input parameter of the `MakeEventLoop()` function. Pass the objects created at step 9 and 10 to the `ServeConnectionRequests()` function. Objects can be passed using smart pointers (`std::shared_ptr`) or ordinary pointers. In the first case, the `kosipc` library takes control of the lifetime of objects. In the second case, the responsibility remains with the developer to ensure that the objects created at steps 9 and 10 are not destroyed before the `kosipc::EventLoop` object finishes.

    > After creating and starting loops, there is no need for synchronization between threads in which the loops are executed.

12. Start the loop for dispatching incoming IPC messages in a separate thread by calling the `Run()` method of the `kosipc::EventLoop` object.

13. In the current thread, start the loop for receiving incoming requests for a dynamic IPC channel connection by calling the `Run()` method of the `kosipc::EventLoop` object. Prior to starting this loop, client processes are locked when a proxy object is created while waiting for the server to respond to the connection request.

Example code for combining the set of components and endpoints of the server into a list of `ServiceList`-type objects using the `AddServices()` method:

```
// Create an application object
kosipc::Application app = kosipc::MakeApplicationAutodetect();

// Create and initialize the group_1 object
kosipc::components::Root    group_1;

group_1.component1.endpoint1 = &impl_1;
group_1.component2.endpoint2 = &impl_2;

// Create a loop for dispatching incoming IPC messages
kosipc::EventLoop loop1 = app.MakeEventLoop(ServeDynamicChannel(group_1));

// Create and initialize the group_2 object
kosipc::components::Root    group_2;

group_2.component3.endpoint1 = &impl_3;
group_2.component4.endpoint2 = &impl_4;

// Create a loop for dispatching incoming IPC messages
kosipc::EventLoop loop2 = app.MakeEventLoop(ServeDynamicChannel(group_2));

// Create and initialize the group_3 object
kosipc::components::Root group_3;

group_3.component5.endpoint1 = &impl_5;
```

```
// Create a loop for dispatching incoming IPC messages
kosipc::EventLoop loop3 = app.MakeEventLoop(ServeDynamicChannel(group_3));

// Create a list of objects
ServiceList endpoints;
endpoints.AddServices(group_1);
endpoints.AddServices(group_2);
endpoints.AddServices(group_3.component5.endpoint1, "SomeCustomEndpointName");

// Create an object that implements publication of endpoints on the server
kosipc::DcmServicePublisher         publisher(app, endpoints);

// Create an object that implements the handler for receiving incoming requests
// to dynamically create an IPC channel
kosipc::SimpleConnectionAcceptor    acceptor(app, endpoints);

// Create an object that implements a loop for receiving incoming requests
// to dynamically create an IPC channel
kosipc::EventLoop   loopDynamicChannel = app.MakeEventLoop(ServeConnectionRequests(&acceptor, &publisher));
```

Working with KPA packages

A *KPA package* is a file in the proprietary KPA format that serves as packaging for a program intended for installation in a KasperskyOS-based solution. It includes the following elements:

- KPA package header. Unique sequence of bytes used to identify the KPA format.
- KPA package manifest. Data structure that is described in a JSON file containing detailed information about the KPA package.
- KPA package components. Aligned byte sequences with arbitrary contents. KPA package components may include executable files, libraries, text data, or any other data required for the program to work.
- KPA package index. Data structure that describes the number of KPA package components, their checksums, and their sizes.

A compressed KPA package is a file in KPAC format. KPAC format is a variant of the ZIP format but with the following limitations: no directory hierarchy, no embedded ZIP archives, and a limit on the size and quantity of its archived files. A KPAC file contains a KPA package, its external signature, and an index file. The external signature of a KPA package is a file in the proprietary KCAT format that resides outside of the KPA package file. The external signature protects against spoofing and modifications of the KPA package itself and the index file of the KPA package. The index file of a KPA package is a file in the proprietary KIDX format that is used to check the integrity of the KPA package.

Scenario for working with KPA packages

During the build, applications must be packed into KPA packages using the `CMake` library platform/kpa.

Applications packed into KPA packages are started by using the ServiceManager system program.

All examples in the KasperskyOS Community Edition SDK use the KPA package mechanism.

KasperskyOS Community Edition SDK includes the following resources for KPA package management:

- `CMake` library platform/kpa intended for building KPA packages during the KasperskyOS-based solution build process. When using functions of the `CMake` library platform/kpa, the KPA package manifest is created automatically.
- Tools that let you build a KPA package from program source files in the system where the SDK is installed and then install the KPA package in a built KasperskyOS-based solution image.
- PackageManager component that lets you install KPA packages in an operational KasperskyOS-based solution, remove KPA packages, and receive information about them.

KPA package manifest

A *KPA package manifest* is a JSON file that contains the information required when installing and using a KPA package. A list of the main keys for the KPA package manifest is presented in the table below.

Main keys of a KPA package manifest

Key name	Value type	Description	Required	Example
`version`	String	Manifest version number in the format `{major}.{minor}`, where `major` is the major version of the manifest and `minor` is the minor version of the manifest. The current version of the manifest is 2.0. It is not recommended to change this version because any change could result in problems with program operation.	Yes	`"version": "2.0"`
`digestType`	String	Identifier (OID) of the hash function that is used to calculate the checksum of object files and other checksums. If the value is not defined or is an empty string, the value `"2.16.840.1.101.3.4.2.1"` (SHA256 hash function) is used by default.	No	`"digestType": "2.16.840.1.101.3.4.2.1"`
`application`	Object	Information about the program.	Yes	Application object
`platform`	Object	Information about the target platform of the program.	Yes	Platform object
`components`	List of objects	List of KPA package components.	No	List of "components" objects
`runConfiguration`	List of objects	List of program startup configurations.	No	List of runConfiguration objects
`privateStorage`	Object	Isolated storage of program data.	No	PrivateStorage object
`extensions`	Object	User-defined object that lets you add custom contents to the manifest to meet the specific demands of the program developer.	No	Custom contents

Application object

The `application` object includes keys containing information about the program installed from the KPA package. A list of these keys is provided in the table below.

List of keys of an application object

Key name	Value type	Description	Required	Example
`id`	String	Unique ID of the program. This ID enables unequivocal identification of installed program. After it is published, the program ID must not change. If the program ID changes, all other participants of the installed program life cycle will consider it as a different program.	Yes	`"id": "helloworld"`
`name`	String	Displayed named of the program (for example, used for display in user interfaces).	Yes	`"name": "Hello World"`
`version`	String	Program version.	Yes	`"version": "1.2.34"`
`buildNumber`	String	Program build number.	Yes	`"buildNumber": "182"`
`systemApplication`	Logical	Logical value indicating whether the program is a system program: `true` means that it is a system program, and `false` means that it is application. In the current version of KasperskyOS Community Edition SDK, the only available value is `false`.	Yes	`"systemApplication": false`
`description`	String	Program description.	No	`"description": "Sample application"`

Platform object

The `platform` object includes keys containing information about the target platform of the program. A list of these keys is provided in the table below.

List of keys of a platform object

Key name		Value type	Description	Required	Example
`id`		String	Operating system ID.	Yes	`"id": "kos.ce"`
`sdk`		Object	Target API level that must be provided by the KasperskyOS Community Edition SDK device for optimal operation of the program, and the name and version of the SDK used to build this program.	Yes	`"sdk": { "buildSdkName": "KasperskyOS-Community-Edition-RaspberryPi4b" "buildSdkVersion": "1.3.0.3" "targetApiLevel": 4 }`
	`buildSdkName`	String	Name of the SDK used to build this program.	Yes	
	`buildSdkVersion`	String	Version of the SDK used to build this program.	Yes	
	`targetApiLevel`	Integer	Target API level that must be provided by the device hosting KasperskyOS Community Edition SDK.	Yes	
`hwSpecification`		Object	Technical specification of the hardware platform.	Yes	`"hwSpecification": { "arch": "aarch64-kos" "cpu": "bcm2711" }`
	`arch`	String	Architecture of the hardware platform.	Yes	
	`cpu`	String	Name of the processor used to run the hardware platform.	Yes	

List of "components" objects

The list of `components` objects includes keys containing information about the components added to the KPA package. A list of these keys is provided in the table below.

List of keys for describing a component instance in the list of components objects

Key name	Value type	Description	Required	Example
`name`	String	Name of the KPA package component.	Yes	`"name": "imageHighResolution"`
`directoryPath`	String	Path of the directory relative to the path `/<package name>/res` where KPA package component should be installed. It is ignored if the `componentType` key value is set to `bin` or `lib`.	No	`"directoryPath": "images/highResolution"`
`digest`	String	Checksum of the KPA package component file.	Yes	`"digest": "2d541fe063c195a3b8a90204f2c234e1b5daf664c`
`componentType`	String	Component type: • `bin` – an executable file. It is put into the `/<program_name>/bin` directory when the program is installed. • `lib` – shared library. It is put into the directory `/<program_name>/lib` when the program is installed. • `res` – user-defined resource. If the `directoryPath` key value is defined, the resource is put into the directory `/<package_name>/res/<directoryPath>` when the program is installed. Otherwise, the resource is put into the directory `/<package_name>/res`. • `manifestLocale` – file containing information for KPA package manifest localization. When the program is installed, KPA package manifest localization files are put into the directory `/<program_name>/manifest_locales` if a value is not defined for the `directoryPath` key.	Yes	`"componentType": "res"`

List of runConfiguration objects

The list of `runConfiguration` objects includes keys containing information about the possible startup configurations of the program. A list of these keys is provided in the table below.

List of keys for describing a startup configuration instance for the list of runConfiguration objects

Key name	Value type	Description	Required	Example
`id`	String	Startup configuration ID that is unique within the specific KPA package.	Yes	`"id": "app"`

`name`	String	Startup configuration name.
		This is a localized string.
			Yes	`"name": "application"`
				or
				`"name": "@runConfigurationLocalizedName@"`

`type`	String	Startup configuration type: • `gui` – process with a graphical user interface. • `service` – endpoint process.	Yes	`"type": "service"`
`args`	List of strings	List of arguments in the form of a string array.	No	`"args": [ "networkSpeed=4096", "nthreads=2" ]`

Key name		Value type	Description	Required	Example
`envVariables`		List of objects	List of environment variables.	No	`"envVariables": { "name": "IMAGES" "value": "images" }`
	`name`	String	Environment variable name.	Yes (if `envVariables` is used)	
	`value`	String	Environment variable value (can be an empty string).	Yes (if `envVariables` is used)	
`primary`		Logical	Indicates whether the startup configuration is the primary one during program startup: `true` means that it is primary, and `false` means that it is not.	Yes	`"primary": true`
`eiid`		String	Program security class. Required for the KasperskyOS security module.	No	`"eiid": "application.Application"`
`path`		String	Path to the KPA package component file. The path is defined relative to `/<application_id>`.	Yes	`"path": "bin/application"`
`usesService`		List of objects	List of startup configurations on which this startup configuration depends. Not supported in the current version of KasperskyOS Community Edition SDK.	No	`"usesService": { "name": "database" "useType": "child" }`
	`name`	String	Name of the utilized startup configuration. If the startup configuration is in another KPA package, the value of this field must have the format `"<package_name>: <startup_configuration_name>"`. If the startup configuration is in this KPA package, the field may have a value in the format `"<package_name>: <startup_configuration_name>"` (`<package_name>` corresponds to the current package) or only `"<startup_configuration_name>"`.	Yes (if `usesService` is used)	
	`useType`	String	Startup configuration usage type: • `child` – the program can start the specified startup configuration as a child process. • `serviceCanUse` – the program may require the specified endpoint. • `serviceActive` – the specified endpoint must be active when the program is started.	Yes (if `usesService` is used)	

PrivateStorage object

The `privateStorage` object includes keys containing information about program data storage. A list of these keys is provided in the table below.

The `privateStorage` object is not supported in the current version of KasperskyOS Community Edition SDK.

List of keys of a privateStorage object

Key name	Value type	Description	Required	Example
`size`	String	Program data storage size in MB.	Yes	`"size": "512"`
`fsType`	String	Program data storage file system type.	Yes	`"fsType": "ext4"`

Tools for managing KPA packages

KasperskyOS Community Edition SDK includes the following tools for managing KPA packages:

- cas-pack for building a KPA package in a system where the SDK is installed.
- cas-inspect for getting information about KPA package contents when working with the SDK.
- cas-pm for installing one or more KPA packages in a built KasperskyOS-based solution image.

The executable files of these tools are located in the `/opt/KasperskyOS-Community-Edition-<platform>-<version>/toolchain/bin/` directory.

cas-pack tool

KasperskyOS Community Edition SDK includes the `cas-pack` tool (the `toolchain/bin/cas-pack` executable file) intended for building a KPA package in a system where the KasperskyOS Community Edition SDK is installed.

Syntax of the shell command for running the `cas-pack` tool:

```
cas-pack {-o|--output} <FILE> --manifest <FILE> --verify [--version] [-h|--help] <FILES>…
```

Parameters:

- **`{-o|--output}`** **`<FILE>`**

  Full name of the built KPA package file.

- `--manifest` `<FILE>`

  Full name of the KPA package manifest file.

- `<FILES>`

  List of the full names of files that will be included in the KPA package. Use a space to separate list items. You can use the `*` character to select all files in the directory.

- `--verify`

  Verification of the presence of all KPA package components specified in its manifest and the absence of unspecified components, and calculation of the checksums of KPA package components and their comparison with those specified in the KPA package manifest.

- `--version`

  Tool version.

- `-h|--help`

  Help text.

Example of the shell command for running the `cas-pack` tool:

```
# Packages all program source files residing in the
# files directory into the KPA package named helloworld.kpa. A KPA package is built
# using information from the manifest.json file, which is the
# KPA package manifest. The built KPA package is verified.

/opt/KasperskyOS-Community-Edition-<platform>-<version>/toolchain/bin/cas-pack
         --output   ./helloworld.kpa
         --manifest ./manifest.json
         --verify
         ./files/*
```

cas-inspect tool

The KasperskyOS Community Edition SDK is delivered with the `cas-inspect` tool (`toolchain/bin/cas-inspect` executable file), which lets you get information about the contents of KPA package when working with SDK.

Syntax of the shell command for running the `cas-inspect` tool:

```
cas-inspect [-h|--help] [--version] {-i|--input} <PACKAGE> --verify
[<COMMAND>] [-o <path>]
```

Parameters:

- **`{-i|--input}`** `<PACKAGE>`

  Path to the KPA package (`*.kpa` file).

- `<COMMAND>`

  Commands:

  - `dump` – prints to standard output the KPA package manifest and information about KPA package components, which includes the size in bytes (`Size`), offset in bytes (`Offset` – relative to the end of the KPA package manifest, and `Absolute` – relative to the start of the KPA package), and the checksum (`Digest`). This parameter is applied by default.

  - `read <manifest|blobs|<hash>` – prints the KPA package manifest (`read manifest`), the contents of all KPA package components (`read blobs`), or the contents of one KPA package component with the defined checksum (`read <hash>`). When using the `-o <path>` parameter, the output is printed to a file. Otherwise, it is printed to the standard output.

  - `list` – prints the checksum, offset (in bytes) relative to the start of the KPA package, and the size (in bytes) for all KPA package components to the standard output.

  - `read-files <FILES>...` – prints the contents of a KPA package component based on the component file name. You can specify multiple names of KPA package component files if you separate these names with a space. When using the `-o <path>` parameter, the output is printed to a file.

  - `list-files` – prints to standard output a list of all names of KPA package component files included in the KPA package manifest.

- `-o <path>`

  Path to the file or directory for saving data when using the commands `read {manifest|blobs|<hash>}` and `read-files <FILES>…`. When printing the KPA package manifest (`read manifest`) or the contents of the KPA package component with the defined checksum (`read <hash>`), you must specify the file path. When printing the contents of all program components (`read blobs`), you must specify the path to the directory where each program component will be saved in a separate file with a name matching the checksum of the specific component. When printing the contents of all KPA package components (`read-files <FILES>…`), you must specify the path to the directory where each KPA package component will be saved in a separate file with the name of the specific component.

- `--verify`

  Verification of the presence of all KPA package components specified in its manifest and the absence of unspecified KPA package components, and calculation of the checksums of KPA package components and their comparison with those specified in the KPA package manifest.

- `-h|--help`

  Help text.

- `--version`

  Tool version.

Examples of shell commands for running the `cas-inspect` tool:

```
# Prints the KPA package manifest and information about
# KPA package components.
cas-inspect -i helloworld.kpa

# Prints the KPA package manifest and information about
# KPA package components to the console, and verifies the availability of the KPA package components
# specified in the manifest and the checksums of
# KPA package components.
cas-inspect -i helloworld.kpa --verify

# Prints the KPA package manifest to a file.
cas-inspect -i helloworld.kpa read manifest -o ./manifest

# Prints the contents of the KPA package component with the defined
# checksum to a file.
cas-inspect -i helloworld.kpa read 5d8071308518a7bb003aa084fc995
d2f09b79e9e52f8cd296cb3ee2644ad3951 -o ./comp

# Prints the contents of each KPA package component to a separate file.
cas-inspect -i helloworld.kpa read blobs -o .

# Prints information about KPA package components to the console.
cas-inspect -i helloworld.kpa list
```

cas-pm tool

KasperskyOS Community Edition SDK includes the `cas-pm` tool (the `toolchain/bin/cas-pm` executable file), which installs KPA packages in a built KasperskyOS-based solution image.

Syntax of the shell command for running the `cas-pm` tool:

```
cas-pm {-p|--pkgsdir} <DIR> {-a|--appsdir} <DIR> [--rootdir <DIR>] [{-l|--layout} <PATH>]
{-e|--extention} <ARG> {-r|--reinstall} <-v[v...]> [--sign-ext <ARG>] [--index-ext <ARG>]
<PACKAGES>… [--version] [-h|--help]
```

Parameters:

- `{-p|--pkgsdir} <DIR>`

  SDK host system path to the directory containing the KPA packages to be installed.

- `{-a|--appsdir} <DIR>`

  SDK host system path to the directory intended for storing KPA packages before they are written to the KasperskyOS-based solution image.

- `--rootdir <DIR>`

  Relative directory that will be used for installing KPA packages to the KasperskyOS-based solution image. Specify the directory in the file system that will be put into the KasperskyOS-based solution image. Information about the location of KPA packages will be entered into the database and will be required by the PackageManager component when removing KPA packages.

- `{-l|--layout} <PATH>`

  Full name of the JSON file that is used to redefine the paths for installing KPA package components. Specify the full file name in the system where the SDK is installed. By default, when a KPA package is installed, its components are put into directories depending on the specific type of KPA package component (for more details, see the `componentType` key in the article titled List of "components" objects). To change the names of the default directories, define your own values for the keys: `bin`, `res`, `lib` and `manifestLocale`. To ensure that the PackageManager component can detect KPA package components after the KasperskyOS-based solution is started, the name of this file must be passed in the `CUSTOM_LAYOUT` parameter of the CMake command `create_package_manager_entity()` (for more details, see PackageManager component usage scenario).

  Example of a `custom_layout_schema.json` file:

  ```
  {
      "bin"            : "custom-bin-path",
      "res"            : "CustomResPath",
      "lib"            : "CustomLibPath",
      "manifestLocale" : "Custom_manifestLocale_Path"
  }
  ```

- `{-e|--extention} <ARG>`

  Extension for a KPA package file. The default value is `kpa`.

- `{-r|--reinstall}`

  Reinstallation of KPA packages.

- `-v[v...]`

  Log level for actions performed by the tool. The number of `v` characters indicates the log level. Messages are printed to standard output. Available values:

  - `-v`

    This level logs non-detailed information about normal operation of the tool, and errors and warnings about potential problems.

  - `-vv[v...]`

    This level adds detailed logging of tool operation information that may be useful to developers for troubleshooting.

- `--sign-ext <ARG>`

  Extension for a KPA package external signature file. For more details about a KPA package external signature, see Working with KPA packages.

- `--index-ext <ARG>`

  Extension for a KPA package index file. For more details about a KPA package index file, see Working with KPA packages.

- `<PACKAGES>`

  List of full names of installed KPA packages in the system where the SDK is installed. You do not need to specify the file extension. Use a space to separate list items.

- `--version`

  Tool version.

- `-h|--help`

  Help text.

Examples of shell commands for running the `cas-pm` tool:

```
# Reinstall the helloworld.kpa package located in the directory defined by the ${PKG_DIR} variable.
# The package will first be put into the directory ${ROOTFS_DIR}/package before it is written to the image of the
# KasperskyOS-based solution. The directory ${ROOTFS_DIR}/package resides in the file system
# that will be copied to the solution image.During installation, detailed logging
# of actions performed by the tool is conducted.
/opt/KasperskyOS-Community-Edition-<platform>-<version>/toolchain/bin/cas-pm
                            --pkgsdir   ${PKG_DIR}
                            --reinstall
                            -vvv
                            --appsdir   ${ROOTFS_DIR}/package
                            --rootdir   ${ROOTFS_DIR}/package
                            --extension kpa
                             ${PKG_DIR}/helloworld
```

PackageManager component usage scenario

The PackageManager component provides an API for managing KPA packages in solutions that are based on KasperskyOS.

The PackageManager component API is built on top of IPC and helps simplify program development. PackageManager is a separate system program that is accessed through IPC. However, developers are provided with a client library that eliminates the necessity of directly using IPC calls.

The programming interface of the PackageManager component is described in the article titled "PackageManager component".

Adding the PackageManager component to a KasperskyOS-based solution

Hereinafter the "client" refers to the program that uses the PackageManager component API to manage KPA packages.

The ExecutionManager system program is supplied as part of the KasperskyOS boot image and starts automatically when the solution is started.

The typical usage scenario for the PackageManager component includes the following steps:

1. Link the client executable file to the client proxy library of PackageManager by adding the following command to the `CMakeLists.txt` file for building the client:

   ```
   target_link_libraries (<name of the CMake target for building the client>
   package_manager::package_manager_proxy)
   ```

2. Add permissions for the necessary events to the solution security policy description:

   a. To enable the PackageManager program to manage KPA packages, the solution security policy must allow the following interactions for the `package_manager.PkgMgrEntity` process class:

   - Access to all endpoints of the `VFS` program.
   - Access to the core endpoints `Sync`, `VMM`, `Thread`, `HAL`, `Handle`, `FS`, `Notice`, `CM` and `Profiler` (their descriptions are located in the directory `sysroot-*-kos/include/kl/core` from the SDK).

   b. To enable a client to call the PackageManager program, the solution security policy must allow the following interactions for the client process class:

   - Access to the appropriate endpoints of the PackageManager program (their descriptions are located in the directory `sysroot-*-kos/include/kl/package_manager` from the SDK).

3. Use of the PackageManager program API in the client code.

   Use the header file `component/package_manager/kos_ipc/package_manager_proxy.h` for this. For more details, refer to PackageManager component.

Using the command line of the KasperskyOS kernel

The KasperskyOS kernel can receive a command line during startup, which lets you, for example, activate the kernel functionality that is disabled by default, configure kernel mechanisms, and specify which hardware I/O interfaces will be used by the kernel on a specific hardware platform.

The kernel command line is one or more space-separated kernel startup parameters:

```
<parameter_name1>[=<parameter_value1>] [<parameter_name2>[=<parameter_value2>]]...
```

Kernel startup parameters are shown in the table below.

To pass the command line to the kernel when starting a solution in QEMU, you need to specify the kernel startup parameters in the `KERNEL_CMDLINE` parameter of the CMake command `generate_qemu_target ()`.

Example of using the `KERNEL_CMDLINE` parameter of the CMake command `generate_qemu_target ()`:

```
generate_qemu_target (qemu
                      ...
                      KERNEL_CMDLINE "gdbstub.connection.timeout=10 gdbstub")
```

To pass the command line to the kernel when starting a solution on the hardware platform, you need to specify the kernel startup parameters in the `BOOTARGS` parameter of the CMake command `build_sd_image ()`:

Example of using the `BOOTARGS` parameter of the CMake command `build_sd_image ()`:

```
build_sd_image (sd-image
                ...
                BOOTARGS "gdbstub.connection.timeout=10 gdbstub")
```

KasperskyOS kernel startup parameters

Parameter	Parameter description
`gdbstub`	Starts the kernel GDB server.

`gdbstub.transport=uart:<UART ID>[?baudrate=<baud rate>]`	Specifies the UART used by the kernel GDB server.
	
	By default, the kernel GDB server uses UART5 on the Raspberry Pi 4 B. To specify a different UART, you need to specify the UART ID in Device Tree Source (DTS) format. For example, to enable the kernel GDB server to use the UART3, you must specify the following parameter:
	
	`gdbstub.transport=uart:/soc/serial@7e201600`
	
	For Raspberry Pi 4 B with the U-Boot bootloader, you also need to make changes to the `config.txt` configuration file. For example, to enable the kernel GDB server to use the UART3, you must specify the following parameter:
	
	`dtoverlay=uart3`
	
	In addition, you need to set the clock rate for the specified UART. To do so, you can use the U-boot bootloader command `fdt set`. For example, to set UART3 to a clock rate of 48 MHz, use the following command:
	
	`fdt set /soc/serial@7e201600 kos,clock-frequency <48000000>`
	
	On the Raspberry Pi 4 B, the UART used by the kernel GDB server has a default baud rate of 115,200. This value can be changed if necessary. For example, the following parameter sets for UART3 the baud rate to 9600:
	
	`gdbstub.transport=uart:/soc/serial@7e201600?baudrate=9600`
	
	When debugging on QEMU, the build system automatically configures this parameter so that the kernel GDB server uses UART1. This corresponds to the QEMU startup parameters (`-serial`) that are also automatically generated by the build system.

Parameter	Parameter description

`gdbstub.connection.timeout=<parameter value>`	Defines the amount of time that the GDB server waits for the GDB debugger to connect when starting the kernel.
	
	When starting the kernel, the GDB server may wait a specific amount of time or may wait indefinitely for the GDB debugger to connect, depending on the parameter value. Parameter can take the following values:
	
	• `-1` – infinite timeout
	• `0` – no waiting
	• `1,2,...,1200` – timeout (in seconds)
	
	For example, to set the timeout to 10 seconds, you must specify the following parameter in the command line:
	
	`gdbstub.connection.timeout=10`
	
	If the `gdbstub.connection.timeout=<parameter value>` parameter is not specified in the kernel command line, the default value of this parameter will be set, which can be obtained from the diagnostic output executed by the kernel. (If the parameter is not specified in the kernel command line, the diagnostic output contains the default parameter value, otherwise it contains the defined parameter value.)
	
	Examples of diagnostic output with a parameter value:
	
	`[KDBG ] GDB connection timeout: 10 second(s).`
	
	`[KDBG ] GDB connection timeout: infinite.`

Parameter	Parameter description

`earlycon={<UART description>|none}`	Defines the UART used by the kernel for diagnostic output.
	
	By default, the kernel uses UART0 for diagnostic output on the Raspberry Pi 4 B.
	
	The UART description has the same syntax as the `earlycon=<UART description>` parameter of the Linux kernel. The following UARTs are supported: 16550, PL011. For example, to enable the kernel to use UART3 for diagnostic output on the Raspberry Pi 4 B, you must specify the following parameter:
	
	`earlycon=pl011,mmio32,0xfe201600`
	
	For Raspberry Pi 4 B with the U-Boot bootloader, you also need to make changes to the `config.txt` configuration file. For example, to enable the kernel GDB server to use the UART3, you must specify the following parameter:
	
	`dtoverlay=uart3`
	
	On the Raspberry Pi 4 B, the UART that is used by the kernel for diagnostic output is set to 115200 baud by default. For UART 16550, you can change the symbol rate by using the `options` parameter in the following construct:
	
	`earlycon=uart,<...><addr>[,<options>[,<*>]]`
	
	For example, the following parameter sets the symbol rate for UART1 on the Raspberry Pi 4 B to 9600 baud:
	
	`earlycon=uart,mmio32,0xfe215040,9600n8`
	
	If none of the UARTs on the Raspberry Pi 4 B are needed for diagnostic output, the following parameter must be specified:
	
	`earlycon=none`
	
	When the solution is run on QEMU, the build system automatically configures this parameter so that the kernel uses UART0 for diagnostic output. This corresponds to the QEMU startup parameters (`-serial`) that are also automatically generated by the build system.
	
	The solution can use the LogRR system program, which receives diagnostic data from other solution programs and the kernel for subsequent output via UART or by saving it to a file. Solution programs relay diagnostic data to the LogRR program by using the client library of this program. At the same time, solution programs can relay diagnostic data to the kernel by using the trace.h API, for example. The LogRR program queries the kernel to stop diagnostic output and then reads diagnostic data from the kernel, including messages from both the kernel and the solution programs. For the LogRR program, one of the possible diagnostic output options is output via the UART using the UART driver operating in user mode. When using this option, even if the `earlycon=none` parameter is passed to the kernel, the diagnostic output will be executed but only after the LogRR program and the UART driver are started. However, if you need to receive diagnostic data before they are started, you need to pass the `earlycon=<UART description>` parameter to the kernel. In this case, before the LogRR program and UART driver are started, diagnostic output is executed by the kernel and then by the LogRR program itself. In this case, the UART driver can use the same UART that was first used by the kernel for diagnostic output.

`nokaslr`	Disables KASLR support.
`handle.disable_detect_invalid_access`	Disables the handle sanitizer functionality for detecting the use of invalid handles.

Parameter	Parameter description
`vmm.overcommit={disabled|exception}`	Configures how dynamically allocated virtual memory is committed. For more information about committing virtual memory, see "Allocating and freeing memory". Parameter takes the following values: • `disabled` – prohibit committing virtual memory in lazy mode. Calls of functions that must set the "lazy" commit mode for a virtual memory region either commit that region and allocate the entire amount of necessary physical memory or end with an error if the required amount of physical memory cannot be allocated. By default, virtual memory can be committed in lazy mode. • `exception` – generate an exception in case of insufficient physical memory when committing virtual memory in lazy mode. If there is insufficient physical memory and the thread queries virtual memory that must be committed in lazy mode, an exception of the `EXCEPTION_OVERCOMMIT` type is thrown in this thread. (The `EXCEPTION_OVERCOMMIT` constant is defined in the header file `sysroot-*-kos/include/hal/exc_codes.h` from KasperskyOS Community Edition SDK.) By default, no exception is thrown, and the thread is locked while waiting for the necessary amount of physical memory to be freed.
`vmm.zero_dedup=0`	Disables zero page deduplication allocated for committing virtual memory in "lazy" mode.
`vmm.zero_dedup.period=<deduplication period>`	Specifies the period (in seconds) for zero page deduplication allocated for committing virtual memory in "lazy" mode. This period is 10 seconds by default. You cannot specify `0`.

KasperskyOS API

Return codes

A KasperskyOS-based solution uses several different types of return codes to encode the results of API functions. Almost all API functions of the `libkos` library that transmit return codes return values of the `Retcode` type, which is defined in the header file `sysroot-*-kos/include/rtl/retcode.h` from the KasperskyOS SDK. Also, return codes of this type are transmitted by API functions of the `kdf` library and API functions of programs supplied as part of the KasperskyOS SDK (for example, drivers and other system programs created for use in KasperskyOS-based solutions). Transport code functions, several API functions of the `libkos` library for working with IPC transport, and functions of some other IPC-related APIs (for example, functions defined in the header file `sysroot-*-kos/include/nk/transport.h` from the KasperskyOS SDK) transmit return codes of the `nk_err_t` type, which is defined in the header file `sysroot-*-kos/include/nk/types.h` from the KasperskyOS SDK. POSIX interface functions and third-party API functions transmit other types of return codes.

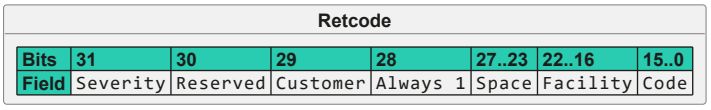
It is recommended that the solution developer encode the results of executing their functions by using return codes of the `Retcode` type.

Retcode return code format

The `Retcode` type is defined in the header file `retcode.h` as a 32-bit signed integer:

```
typedef __INT32_TYPE__ Retcode;
```

The set of return codes of the `Retcode` type consists of a success code with a value of `0` and error codes. The error code is interpreted as a data structure whose format is shown in the figure below.



Retcode return code format

This format provides for multiple fields that contain not only information about the results of a function execution (in the `Code` field), but also the following additional information:

- Flag in the `Severity` field that distinguishes the error code (`1`) from the success code (`0`). This field is used for backward compatibility with other error code formats.
- Flag in the `Customer` field indicating that the error code was defined by the developers of the solution and not by the developers of software from the KasperskyOS SDK.

  Thanks to the flag in the `Customer` field, developers of a solution and developers of software from the KasperskyOS SDK can define error codes from non-overlapping sets.
- Global ID of the error code in the `Space` field.

Global IDs let you define non-overlapping sets of error codes. Error codes can be generic or specific. Generic error codes can be used in the APIs of any solution components and in the APIs of any constituent parts of solution components (for example, a driver or VFS may be a constituent part of a solution component). Specific error codes are used in the APIs of one or more solution components or in the APIs of one or more constituent parts of solution components.

For example, the `RC_SPACE_GENERAL` ID corresponds to generic errors, the `RC_SPACE_KERNEL` ID corresponds to error codes of the kernel, and the `RC_SPACE_DRIVERS` ID corresponds to error codes of drivers.

- Local ID of the error code in the `Facility` field.

  Local IDs let you define non-overlapping subsets of error codes within the set of error codes corresponding to one global ID. For example, the set of error codes with the global ID `RC_SPACE_DRIVERS` includes non-overlapping subsets of error codes with the local IDs `RC_FACILITY_I2C`, `RC_FACILITY_USB`, and `RC_FACILITY_BLKDEV`. (These local identifiers are defined in the header file `sysroot-*-kos/include/kdf/facilities.h` from the KasperskyOS SDK.)

The global and local IDs of specific error codes are assigned by the developers of a solution and by the developers of software from the KasperskyOS SDK independently of each other. In other words, two sets of global IDs are generated. Each global ID has a unique meaning within one set. Each local ID has a unique meaning within a set of local IDs related to one global ID.

This type of centralized approach helps avoid situations in which the same error codes have various meanings within a solution. This is necessary to eliminate a potential problem transmitting error codes through different APIs. For example, this problem occurs when drivers call `kdf` library functions, receive error codes, and return these codes through their own APIs. If error codes are generated without a centralized approach, the same error code can have different meanings for the `kdf` library and for the driver. Under these conditions, drivers return correct error codes only if the error codes of the `kdf` library are converted into error codes of each driver. In other words, error codes are assigned in such way that does not require conversion of these codes during their transit through various APIs.

General return codes of the Retcode type

The general return codes of the `Retcode` type as defined in the header file `retcode.h` are shown in the table below. This table also shows the mapping between the return codes of the `Retcode` type and the return codes of the `nk_err_t` type, which are defined in the header file `sysroot-*-kos/include/nk/errno.h` from the KasperskyOS SDK. This mapping is required for conversions between these two types of return codes. For example, conversion is required if a developed function that returns values of the `Retcode` type "wraps" a transport code function that returns values of the `nk_err_t` type. However, please keep in mind that conversion is not always possible for the following reasons:

- Some return codes are not mapped to return codes of the other type.
- Some return codes of the `nk_err_t` type are mapped to multiple return codes of the `Retcode` type. In other words, it may be impossible to explicitly determine which return code of the `Retcode` type corresponds to a return code of the `nk_err_t` type that needs to be converted.

General return codes of the Retcode type

Retcode return code	Description	Nk_err_t return code
`rcOk` (corresponds to the `0` value)	The function completed successfully.	`NK_EOK`
`rcInvalidArgument`	Invalid function parameter.	`NK_EINVAL`

Retcode return code	Description	Nk_err_t return code
rcNotConnected	No connection between the client and server sides of interaction. For example, there is no server IPC handle.	NK_ENOTCONN
rcOutOfMemory	Insufficient memory to perform the operation.	NK_ENOMEM
rcBufferTooSmall	Insufficient buffer size.	NK_EBADMSG
rcInternalError	The function ended with an internal error related to incorrect logic. Some examples of internal errors include values outside of the permissible limits, and null indicators and values where they are not permitted.	No
rcTransferError	Error sending an IPC message.	No
rcReceiveError	Error receiving an IPC message.	No
rcSourceFault	IPC message was not transmitted due to the IPC message source.	No
rcTargetFault	IPC message was not transmitted due to the IPC message recipient.	No
rcIpcInterrupt	IPC was interrupted by another process thread.	No
rcInvalidFormat	Data format is incorrect.	No
rcRestart	Indicates that the function needs to be called again.	No
rcFail	The function ended with an error.	No
rcNoCapability	The operation cannot be performed with the resource.	NK_ENOENT
rcNotReady	Initialization failed.	NK_EAGAIN
rcUnimplemented	The function was not implemented.	No
rcBufferTooLarge	Large buffer size.	NK_EBADMSG
rcBusy	Resource temporarily unavailable.	No
rcResourceNotFound	Resource not found.	No
rcTimeout	Timed out.	NK_EAGAIN
rcSecurityDisallow	The operation was denied by security mechanisms.	NK_EPERM
rcFutexWouldBlock	The operation will result in a block.	No
rcAbort	The operation was aborted.	NK_ENOENT
rcInvalidThreadState	Invalid function called in the interrupt handler.	No
rcAlreadyExists	Set of elements already contains the element being added.	NK_ENOTEMPTY
rcInvalidOperation	Operation cannot be completed.	No
rcHandleRevoked	Resource access rights were revoked.	NK_EREVOKED
rcQuotaExceeded	Resource quota exceeded.	No
rcDeviceNotFound	Device not found.	No
rcOverflow	An overflow occurred.	No
rcAlreadyDone	Operation has already been completed.	No
rcNotRun	Operation has not been started.	No
rcEntropySourceStuck	Source of entropy may be unreliable.	No
rcDeadlock	A deadlock has occurred.	No
rcInvalidHandle	Handle is closed.	No

Defining a Retcode error code

To define an error code of the `Retcode` type, the solution developer must use the `MAKE_RETCODE()` macro that is defined in the `retcode.h` header file. The developer must also use the `customer` parameter to pass the symbolic constant `RC_CUSTOMER_TRUE`.

Example:

```
#define LV_EBADREQUEST MAKE_RETCODE(RC_CUSTOMER_TRUE, RC_SPACE_APPS, RC_FACILITY_LogViewer, 5, "Bad request")
```

> An error description that is passed via the `desc` parameter is not used by the `MAKE_RETCODE()` macro. This description is needed to create a database of error codes when building a KasperskyOS-based solution. At present, a mechanism for creating and using such a database has not been implemented.

Reading Retcode type error code structure fields

The macros `RC_GET_CUSTOMER()`, `RC_GET_SPACE()`, `RC_GET_FACILITY()`, and `RC_GET_CODE()` defined in the `retcode.h` header file allow you to read the fields of the `Retcode` error code structure.

The `RETCODE_HR_PARAMS()` and `RETCODE_HR_FMT()` macros defined in the `sysroot-*-kos/include/rtl/retcode_hr.h` header file from the KasperskyOS SDK are used for formatted display of `Retcode` type error details.

libkos library

The `libkos` library is the basic KasperskyOS library that provides the set of APIs that allow programs and other libraries (for example, `libc` and `kdf`) to use core endpoints. The APIs provided by the `libkos` library enable solution developers to do the following:

- Manage processes, threads, and virtual memory.
- Control access to resources.
- Perform input/output operations.
- Create IPC channels.
- Manage power.

- Obtain statistical data on the system.
- Use other capabilities supported by core endpoints.

> On hardware platforms running an Arm processor architecture, input and output parameters of the `libkos` library API cannot be saved in Device memory because this could lead to unpredictable behavior. (Exceptions to this rule are the `addr` parameter of the `KnVmQuery()` function from the vmm_api.h API, the `reg` and `baseReg` parameters of functions from the mmio.h API, the `va` parameter of the `KnHalFlushCache()` function from the hal_api.h API, and the `va` parameter of the `KosCpuCacheFlush()` function from the cpucache.h API.) Parameters of the `libkos` library API must be saved in Normal memory. To copy data from "Device memory" to "Normal memory" and vice versa, you must use the `RtlPedanticMemcpy()` function declared in the header file `sysroot-*-kos/include/rtl/string_pedantic.h` from the KasperskyOS SDK.

Managing handles (handle_api.h)

The API is defined in the `sysroot-*-kos/include/coresrv/handle/handle_api.h` header file from the KasperskyOS SDK.

The API is intended for performing operations with handles. Handles have the `Handle` type, which is defined in the header file `sysroot-*-kos/include/handle/handletype.h` from the KasperskyOS SDK.

Locality of handles

Each process receives handles from its own handle space irrespective of other processes. The handle spaces of different processes are absolutely identical in that they consist of the same set of values. Therefore, a handle is unique (has a unique value) only within the handle space of the single process that owns the particular handle. In other words, different processes may have identical handles that identify different resources, or may have different handles that identify the same resource.

Handle permissions mask

A handle permissions mask has a size of 32 bits and consists of a general part and a specialized part. The general part describes the general rights that are not specific to any particular resource (the flags of these rights are defined in the header file `sysroot-*-kos/include/services/ocap.h` from the KasperskyOS SDK). For example, the general part contains the `OCAP_HANDLE_TRANSFER` flag, which defines the permission to transfer the handle. The specialized part describes the rights that are specific to the particular user resource or system resource. The flags of the specialized part's permissions for system resources are defined in the `ocap.h` header file. The structure of the specialized part for user resources is defined by the resource provider by using the `OCAP_HANDLE_SPEC()` macro that is defined in the `ocap.h` header file. The resource provider must export the public header files describing the flags of the specialized part.

When the handle of a system resource is created, the permissions mask is defined by the KasperskyOS kernel, which applies permissions masks from the `ocap.h` header file. It applies permissions masks with names such as `OCAP_*_FULL` (for example, `OCAP_IOPORT_FULL`, `OCAP_TASK_FULL`, `OCAP_FILE_FULL`) and `OCAP_IPC_*` (for example, `OCAP_IPC_SERVER`, `OCAP_IPC_LISTENER`, `OCAP_IPC_CLIENT`).

When the handle of a user resource is created, the permissions mask is defined by the user.

When a handle is transferred, the permissions mask is defined by the user but the transferred access rights cannot be elevated above the access rights of the process.

Creating handles

Information about API functions is provided in the table below.

Creating handles of system resources

Handles of system resources are created when these resources are created. For example, handles are created when an interrupt or MMIO memory region is registered, and when a DMA buffer, thread, or process is created.

Creating handles of user resources

Handles of user resources are created by the providers of these resources by using the `KnHandleCreateUserObject()` or `KnHandleCreateUserObjectEx()` function.

The context of a user resource must be defined through the `context` parameter. The *user resource context* consists of data that allows the resource provider to identify the resource and its state when access to the resource is requested by other processes. This normally consists of a data set with various types of data (structure). For example, the context of a file may include the name, path, and cursor position. The user resource context is used as the resource transfer context or is used together with multiple resource transfer contexts.

You must use the `rights` parameter to define the handle permissions mask.

Creating IPC handles

An *IPC handle* is a handle that identifies an IPC channel. IPC handles are used to execute system calls. A client IPC handle is necessary for executing a `Call()` system call. A server IPC handle is necessary for executing the `Recv()` and `Reply()` system calls. A *listener handle* is a server IPC handle that has extended rights allowing it to add IPC channels to the set of IPC channels identified by this handle. A *callable handle* is a client IPC handle that simultaneously identifies the IPC channel to a server and an endpoint of this server.

A server creates a callable handle and passes it to a client so that the client can use the server endpoint. The client initializes IPC transport by using the callable handle that it received. In addition, the client specifies the `INVALID_RIID` value as the Runtime Implementation Identifier (RIID) in the proxy object initialization function. To create a callable handle, call the `KnHandleCreateUserObjectEx()` function and specify the server IPC handle and the RIID in the `ipcChannel` and `riid` parameters, respectively. Use the `context` parameter to specify the data to be associated with the callable handle. The server will be able to receive the pointer to this data when dereferencing the callable handle. (Even though the callable handle is an IPC handle, the kernel puts it into the `base_.self` field of the constant part of an IPC request.)

To create the client IPC handle, server IPC handle, and listener IPC handle and associate them with each other, call the `KnHandleConnect()` or `KnHandleConnectEx()` function. These functions are used to statically create IPC channels. The `KnHandleConnect()` function creates IPC handles from the handle space of the calling process. However, the client IPC handle can be transferred to another process. The `KnHandleConnectEx()` function can create IPC handles from the handle space of the calling process or from the handle spaces of other processes, such as the client and server.

When calling the `KnHandleConnect()` or `KnHandleConnectEx()` function with the `INVALID_HANDLE` value in the parameter that defines the listener handle, a new listener handle is created. However, the server IPC handle and listener IPC handle in the output parameters are the same handle. If a listener handle is specified when calling the `KnHandleConnect()` or `KnHandleConnectEx()` function, the created server IPC handle will provide the capability to receive IPC requests over all IPC channels associated with this listener handle. In this case, the server IPC handle and listener IPC handle in the output parameters are different handles. (The first IPC channel associated with the listener handle is created when calling the `KnHandleConnect()` or `KnHandleConnectEx()` function with the `INVALID_HANDLE` value in the parameter that defines the listener handle. The second and subsequent IPC channels associated with the listener handle are created during the second and subsequent calls of the `KnHandleConnect()` or `KnHandleConnectEx()` function specifying the listener handle that was obtained during the first call.)

To call a listener handle that is not associated with a client IPC handle and server IPC handle, call the `KnHandleCreateListener()` function. (The `KnHandleConnect()` and `KnHandleConnectEx()` functions create a listener handle associated with a client IPC handle and server IPC handle.) The `KnHandleCreateListener()` function is convenient for creating a listener handle that will be subsequently bound to callable handles.

To create a client IPC handle for querying the Kaspersky Security Module through the security interface, call the `KnHandleSecurityConnect()` function. This function is called by the `libkos` library when initializing IPC transport for querying the security module.

Information about API functions

handle_api.h functions

Function	Information about the function
`KnHandleCreateUserObject()`	Purpose Creates a handle. Parameters • [in] `type` – handle type. Fictitious parameter that must take a value ranging from the `HANDLE_TYPE_USER_FIRST` constant to the `HANDLE_TYPE_USER_LAST` constant as defined in the header file `sysroot-*-kos/include/handle/handletype.h` from the KasperskyOS SDK. • [in] `rights` – handle permissions mask. • [in,optional] `context` – pointer to the data that should be associated with the handle, or `RTL_NULL` if this association is not required. • [out] `handle` – pointer to the handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KnHandleCreateUserObjectEx()`	Purpose Creates a handle. Parameters • [in] `type` – handle type. Fictitious parameter that must take a value ranging from the `HANDLE_TYPE_USER_FIRST` constant to the `HANDLE_TYPE_USER_LAST` constant as defined in the header file `sysroot-*-kos/include/handle/handletype.h` from the KasperskyOS SDK. • [in] `rights` – handle permissions mask. • [in,optional] `context` – pointer to the data that should be associated with the handle, or `RTL_NULL` if this association is not required. • [in,optional] `ipcChannel` – server IPC handle, or `INVALID_HANDLE` if you do not need to create a callable handle. • [in,optional] `riid` – Runtime Implementation Identifier (RIID), or `INVALID_RIID` if you do not need to create a callable handle. • [out] `handle` – pointer to the handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnHandleConnect()`	Purpose Creates and connects the client, server, and listener IPC handles. Parameters • [in,optional] `ls` – listener handle, or `INVALID_HANDLE` if you need to create it. • [out,optional] `outLs` – pointer to the listener handle. You can specify `RTL_NULL` if the `ls` parameter is used to define the listener handle. • [out,optional] `outSr` – pointer to the server IPC handle, or `RTL_NULL` to not create a server IPC handle if the `ls` parameter is used to define the listener handle. • [out] `outCl` – pointer to the client IPC handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KnHandleConnectEx()`	Purpose Creates and connects the client, server, and listener IPC handles. Parameters • [in] `server` – handle of the server process. • [in,optional] `srListener` – listener handle from the handle space of the server process, or `INVALID_HANDLE` if you need to create it. • [in] `client` – handle of the client process. • [out,optional] `outSrListener` – pointer to the listener handle from the handle space of the server process. You can specify `RTL_NULL` if the `srListener` parameter is used to define the listener handle. • [out,optional] `outSrEndpoint` – pointer to the server IPC handle from the handle space of the server process, or `RTL_NULL` to not create a server IPC handle if the `srListener` parameter is used to define the listener handle. • [out] `outClEndpoint` – pointer to the client IPC handle from the handle space of the client process. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnHandleSecurityConnect()`	Purpose Creates a client IPC handle for querying the Kaspersky Security Module through the security interface. Parameters • [out] `client` – pointer to the handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnHandleCreateListener()`	Purpose Creates a listener handle that is not associated with a client IPC handle and server IPC handle. Parameters • [out] `listener` – pointer to the listener handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Transferring handles

Information about API functions is provided in the table below.

Overview of handle transfers

Handles are transferred between processes so that resource consumers can gain access to required resources. Due to the specific locality of handles, a handle transfer initiates the creation of a handle from the handle space of the recipient process. This handle is registered as a descendant of the transferred handle and identifies the same resource.

One handle can be transferred multiple times to one or more processes. Each transfer initiates the creation of a new descendant of the transferred handle on the recipient process side. A process can transfer handles that it received from other processes or the KasperskyOS kernel. For this reason, a handle may have multiple generations of descendants. The generation hierarchy of handles for each resource is stored in the KasperskyOS kernel in the form of a *handle inheritance tree*.

A process can transfer handles for user resources and system resources if the access rights of these handles permit such a transfer (the `OCAP_HANDLE_TRANSFER` flag is set in the permissions mask). A descendant may have less access rights than an ancestor. For example, a transferring process with read-and-write permissions for a file can transfer read-only permissions. The transferring process can also prohibit the recipient process from further transferring the handle. Access rights are defined in the transferred permissions mask for the handle.

Conditions for transferring handles

To enable processes to transfer handles to other processes, the following conditions must be met:

1. An IPC channel is created between the processes.
2. The solution security policy (`security.psl`) allows interaction between process classes.
3. Interface methods are implemented for transferring handles.

> The API task.h enables a parent process to pass handles to a child process that is not yet running.

In an IDL description, signatures of interface methods for transferring handles have input (`in`) and/or output (`out`) parameters of the `Handle` type or `array` type with elements of the `Handle` type. Up to 255 handles can be passed through the input parameters of one method. This same number of handles can be received through output parameters.

Example IDL description that defines the signatures of interface methods for transferring handles:

```
package IpcTransfer
interface {
    PublishResource1(in Handle handle, out UInt32 result);
    PublishResource7(in Handle handle1, in Handle handle2,
                     in Handle handle3, in Handle handle4,
                     in Handle handle5, in Handle handle6,
                     in Handle handle7, out UInt32 result);
    OpenResource(in UInt32 ID, out Handle handle);
}
```

For each parameter of the `Handle` type, the transport code generator creates a field of the `nk_handle_desc_t` type (hereinafter also referred to as the *handle transport container*) in the `*_req` IPC request structure and/or `*_res` IPC response structure. This type is declared in the header file `sysroot-*-kos/include/nk/types.h` from the KasperskyOS SDK and comprises a structure consisting of the following three fields: `handle` field for the handle, `rights` field for the handle permissions mask, and the `badge` field for the resource transfer context.

Resource transfer context

The *resource transfer context* consists of data that allows the server to identify the resource and its state when access to the resource is requested via descendants of the transferred handle. This normally consists of a data set with various types of data (structure). For example, the transfer context of a file may include the name, path, and cursor position. The server receives a pointer to the resource transfer context when dereferencing a handle.

Regardless of whether or not the server is the resource provider, the server can associate each handle transfer with a separate resource transfer context. This resource transfer context is bound only to the handle descendants (handle inheritance subtree) that were generated as a result of a specific transfer of the handle. This lets you define the state of a resource in relation to a separate transfer of the handle of this resource. For example, for cases when one file may be accessed multiple times, the file transfer context lets you define which specific opening of this file corresponds to a received IPC request.

If the server is the resource provider, each transfer of the handle of this resource is associated with the user resource context by default. In other words, the user resource context is used as the resource transfer context for each handle transfer if the particular transfer is not associated with a separate resource transfer context.

A server that is the resource provider can use both the user resource context and the resource transfer context together. For example, the name, path and size of a file is stored in the user resource context while the cursor position can be stored in multiple resource transfer contexts because each client can work with different parts of the file. Technically, joint use of the user resource context and resource transfer contexts is possible because the resource transfer contexts store a pointer to the user resource context.

If the client uses multiple various-type resources of the server, the resource transfer contexts (or contexts of user resources if they are used as resource transfer contexts) must be typified objects of the `KosObject` type (for details about `KosObject`s, see Using KosObjects (objects.h)). This is necessary so that the server can verify that the client using a resource has sent the interface method the handle of the specific resource that corresponds to this method. This verification is required because the client could mistakenly send the interface method a resource handle that does not correspond to this method. For example, a client may have received a file handle and sent it to an interface method for working with volumes.

To associate a handle transfer with a resource transfer context, the server puts the handle of the resource transfer context object into the `badge` field of the `nk_handle_desc_t` structure. The *resource transfer context object* is the kernel object that stores the pointer to the resource transfer context. To create a resource transfer context object, call the `KnHandleCreateBadge()` function. This function is bound to the notification mechanism because a server needs to know when a resource transfer context object will be closed and deleted. The server needs this information to free up or re-use memory that was allotted for storing the resource transfer context.

The resource transfer context object will be closed upon the closure or revocation of the handle descendants that comprise the handle inheritance subtree whose root node was generated by the transfer of this handle in association with this object. (A transferred handle may be closed intentionally or unintentionally, such as when a recipient client is unexpectedly terminated.) After receiving a notification regarding the closure of a resource transfer context object, the server closes the handle of this object. After this, the resource transfer context object will be deleted. After receiving a notification regarding the deletion of the resource transfer context object, the server frees up or re-uses the memory that was allotted for storing the resource transfer context.

One resource transfer context object can be associated with only one handle transfer.

Packaging data into the transport container of a handle

To package a handle, handle permissions mask, and resource transfer context object handle into a handle transport container, use the `nk_handle_desc()` macro that is defined in the header file `sysroot-*-kos/include/nk/types.h` from the KasperskyOS SDK. This macro receives a variable number of parameters.

If no parameter is passed to the macro, the `NK_INVALID_HANDLE` value will be written to the `handle` field of the `nk_handle_desc_t` structure. If one parameter is passed to the macro, this parameter is interpreted as the handle. If two parameters are passed to the macro, the first parameter is interpreted as the handle and the second parameter is interpreted as the handle permissions mask. If three parameters are passed to the macro, the first parameter is interpreted as the handle, the second parameter is interpreted as the handle permissions mask, and the third parameter is interpreted as the resource transfer context object handle.

Extracting data from the transport container of a handle

To extract the handle, handle permissions mask, and pointer to the resource transfer context from the transport container of a handle, use the `nk_get_handle()`, `nk_get_rights()` and `nk_get_badge_op()` (or `nk_get_badge()`) functions, respectively, which are declared in the header file `sysroot-*-kos/include/nk/types.h` from the KasperskyOS SDK. The `nk_get_badge_op()` and `nk_get_badge()` functions should be used only when dereferencing handles.

Handle transfer scenarios

The scenario for transferring handles from a client to the server includes the following steps:

1. The client packages the handles and handle permissions masks into fields of the `*_req` IPC requests structure of the `nk_handle_desc_t` type.
2. The client calls the interface method for transferring handles to the server. The `Call()` system call is executed when this method is called.
3. The server receives an IPC request by executing the `Recv()` system call.
4. The dispatcher on the server side calls the method corresponding to the IPC request. This method extracts the handles and handle permissions masks from fields of the `*_req` IPC request structure of the `nk_handle_desc_t` type.

The scenario for transferring handles from the server to a client includes the following steps:

1. The client calls the interface method for receiving handles from the server. The `Call()` system call is executed when this method is called.
2. The server receives an IPC request by executing the `Recv()` system call.
3. The dispatcher on the server side calls the method corresponding to the IPC request. This method packages the handles, handle permissions masks and resource transfer context object handles into fields of the `*_res` IPC response structure of the `nk_handle_desc_t` type.

4. The server responds to the IPC request by executing the `Reply()` system call.

5. On the client side, the interface method returns control. After this, the client extracts the handles and handle permissions masks from fields of the `*_res` IPC response structure of the `nk_handle_desc_t` type.

If the transferring process defines more access rights in the transferred handle permissions mask than the access rights defined for the transferred handle (which it owns), the transfer is not completed. In this case, the `Call()` system call executed by the transferring or recipient client or the `Reply()` system call executed by the transferring server ends with the `rcSecurityDisallow` error.

Information about API functions

handle_api.h functions

Function	Information about the function
`KnHandleCreateBadge()`	Purpose Creates a resource transfer context object and configures a notification mechanism for monitoring the life cycle of this object. Parameters • [in] `notice` – identifier of the notification receiver. • [in] `eventId` – identifier of the "resource–event mask" entry in the notification receiver. • [in,optional] `context` – pointer to the data that should be associated with the handle transfer, or `RTL_NULL` if this association is not required. • [out] `handle` – pointer to the handle of the resource transfer context object. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information The notification receiver is configured to receive notifications about events that match the `EVENT_OBJECT_DESTROYED` and `EVENT_BADGE_CLOSED` flags of the event mask.

Duplicating handles

Handle duplication is similar to a handle transfer, but duplication is performed within a process. A handle descendant is created in the same process and from the same handle space. The rights of the handle descendant may be less than or equal to the rights of the original handle. Handle duplication can be associated with a resource transfer context object. This lets you use the notification mechanism to track the closure or revocation of all handle descendants that form the handle inheritance subtree whose root node was generated by the duplication operation. It also provides the capability to revoke these descendants.

To duplicate a handle, call the `KnHandleCopy()` function. To do so, the `OCAP_HANDLE_COPY` flag must be set in the handle permissions mask.

Information about API functions is provided in the table below.

handle_api.h functions

Function	Information about the function
`KnHandleCopy()`	Purpose Duplicates a handle. As a result of duplication, the calling process receives the handle descendant. Parameters • [in] `inHandle` – original handle. • [in] `newRightsMask` – permissions mask of the handle descendant. • [in,optional] `copyBadge` – handle of the resource transfer context object, or `INVALID_HANDLE` if you do not need to associate handle duplication with this object. • [out] `outHandle` – pointer to the handle descendant. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Dereferencing handles

When *dereferencing a handle*, the client sends the handle to the server, and the server receives a pointer to the resource transfer context, the permissions mask of the sent handle, and the ancestor of the handle sent by the client and already owned by the server. Dereferencing occurs when a resource consumer that called methods for working with a resource (such as read/write or access closure) sends the resource provider the handle that was received from this resource provider when access to the resource was opened.

Dereferencing handles requires fulfillment of the same conditions and utilizes the same mechanisms and data types as when transferring handles. A handle dereferencing scenario includes the following steps:

1. The client packages the handle into a field of the `*_req` IPC request structure of the `nk_handle_desc_t` type.
2. The client calls the interface method for sending the handle to the server for the purpose of performing operations with the resource. The `Call()` system call is executed when this method is called.
3. The server receives the IPC request by executing the `Recv()` system call.
4. The dispatcher on the server side calls the method corresponding to the IPC request. This method verifies that the dereferencing operation was specifically executed instead of a handle transfer. Then the called method has the option to verify that the access rights of the dereferenced handle (that was sent by the client) permit the requested actions with the resource, and extracts the pointer to the resource transfer context from the field of the `*_req` request structure of the `nk_handle_desc_t` type.

To perform verification, the server uses the `nk_is_handle_dereferenced()` and `nk_get_badge_op()` functions that are declared in the header file `sysroot-*-kos/include/nk/types.h` from the KasperskyOS SDK.

types.h (snippet)

```
/**
* Returns a value different from null if
* the handle in the transport container of
* "desc" is received as a result of dereferencing
* the handle.Returns null if the handle
* in the transport container of "desc" is received
* as a result of a handle transfer.
```

```
*/
static inline
nk_bool_t nk_is_handle_dereferenced(const nk_handle_desc_t *desc)

/**
* Extracts the pointer to the resource transfer context
* "badge" from the transport container of "desc"
* if the permissions mask that was put in the transport
* container of the desc handle has the operation flags set.
* If successful, the function returns NK_EOK, otherwise it returns an error code.
*/
static inline
nk_err_t nk_get_badge_op(const nk_handle_desc_t *desc,
                         nk_rights_t operation,
                         nk_badge_t *badge)
```

Generally, the server does not require the handle that was received from dereferencing because the server normally retains the handles that it owns, for example, within the contexts of user resources. However, the server can extract this handle from the handle transport container if necessary.

Revoking handles

A process can revoke descendants of a handle that it owns. Handles are revoked according to the handle inheritance tree.

Revoked handles are not closed. However, you cannot query resources via revoked handles. Any function that receives the handle will end with the `rcHandleRevoked` error if the function is called with a revoked handle.

To revoke the descendants of a handle, call the `KnHandleRevoke()`, `KnHandleRevokeEx()`, or `KnHandleRevokeSubtree()` function. The `KnHandleRevokeSubtree()` function uses the resource transfer context object that is created when transferring handles.

If each handle of a system resource in all processes that own these handles is closed or revoked, this system resource will be deleted.

Information about API functions is provided in the table below.

handle_api.h functions

Function	Information about the function
`KnHandleRevoke()`	Purpose Revokes the descendants of the specified handle and closes the specified handle. Parameters • [in] `handle` – a handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KnHandleRevokeEx()`	Purpose Revokes the descendants of the specified handle and (optionally) closes the specified handle. Parameters • [in] `handle` – a handle. • [in] `close` – boolean value that determines whether to close the handle specified in the `handle` parameter (`rtl_true` – yes, `rtl_false` – no). The parameter type is defined in the `sysroot-*-kos/include/rtl/stdbool.h` header file from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnHandleRevokeSubtree()`	Purpose Revokes the handles that make up the inheritance subtree of the specified handle. Parameters • [in] `handle` – a handle. The handles forming the inheritance subtree of this handle are revoked. • [in] `badge` – handle that identifies the resource transfer context object, which defines the inheritance subtree of the handles to revoke. The root node of this subtree is the handle that was generated by the transfer or duplication of the handle that is defined through the `handle` parameter and is associated with the resource transfer context object. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Closing handles

A process can close the handles that it owns. Closing a handle terminates the association between an ID and a resource, thereby releasing the ID. Closing a handle does not invalidate its ancestors and descendants (in contrast to revoking a handle, which actually invalidates the descendants of the handle). In other words, the ancestors and descendants of a closed handle can still be used to provide access to the resource that they identify. Also, closing a handle does not disrupt the handle inheritance tree associated with the resource identified by the particular handle. The place of a closed handle is occupied by its ancestor. In other words, the ancestor of a closed handle becomes the direct ancestor of the descendants of the closed handle.

To close the handle, call the `KnHandleClose()` function.

If each handle of a system resource in all processes that own these handles are revoked (see Revoking handles) or closed, this system resource will be deleted.

Information about API functions is provided in the table below.

handle_api.h functions

Function	Information about the function
`KnHandleClose()`	Purpose Closes a handle. Parameters • [in] `handle` – a handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Getting a security ID (SID)

By getting the SID values for different handles, you can determine whether these handles identify different resources or the same resource.

To get a security ID for a handle, call the `KnHandleGetSidByHandle()` function. To do so, the `OCAP_HANDLE_GET_SID` flag must be set in the handle permissions mask.

Information about API functions is provided in the table below.

handle_api.h functions

Function	Information about the function
`KnHandleGetSidByHandle()`	Purpose Receives a security ID (SID) based on a handle. Parameters • [in] `handle` – a handle. • [out] `sid` – pointer to the security ID (SID). Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

OCap usage example

This example describes an OCap usage scenario in which the resource provider provides the following methods for accessing its resources:

- `OpenResource()` – opens access to the resource.
- `UseResource()` – uses the resource.
- `CloseResource()` – closes access to the resource.

The resource consumer uses these methods.

IDL description:

```
package SimpleOCap
interface {
    OpenResource(in UInt32 ID, out Handle handle);
    UseResource(in Handle handle, in UInt8 param, out UInt8 result);
    CloseResource(in Handle handle);
}
```

The scenario includes the following steps:

1. The resource provider creates the user resource context and calls the `KnHandleCreateUserObject()` function to create the resource handle. The resource provider saves the resource handle in the user resource context.

2. The resource consumer calls the `OpenResource()` method to open access to the resource.

    a. The resource provider creates the resource transfer context and calls the `KnHandleCreateBadge()` function to create a resource transfer context object and configure the notification receiver to receive notifications regarding the closure and deletion of the resource transfer context object. The resource provider saves the handle of the resource transfer context object and the pointer to the user resource context in the resource transfer context.

    b. The resource provider uses the `nk_handle_desc()` macro to package the resource handle, permissions mask of the handle, and pointer to the resource transfer context object into the handle transport container.

    c. The handle is transferred from the resource provider to the resource consumer, which means that the resource consumer receives a descendant of the handle owned by the resource provider.

    d. The `OpenResource()` method call completes successfully. The resource consumer extracts the handle and permissions mask of the handle from the handle transport container by using the `nk_get_handle()` and `nk_get_rights()` functions, respectively. The handle permissions mask is not required by the resource consumer to query the resource, but is transferred so that the resource consumer can find out its permissions for accessing the resource.

3. The resource consumer calls the `UseResource()` method to utilize the resource.

    a. The handle that was received from the resource provider at step 2 is used as a parameter of the `UseResource()` method. Before calling this method, the resource consumer uses the `nk_handle_desc()` macro to package the handle into the handle transport container.

    b. The handle is dereferenced, after which the resource provider receives the pointer to the resource transfer context.

    c. The resource provider uses the `nk_is_handle_dereferenced()` function to verify that the dereferencing operation was completed instead of a handle transfer.

    d. The resource provider verifies that the access rights of the dereferenced handle (that was sent by the resource consumer) allows the requested operation with the resource, and extracts the pointer to the resource transfer context from the handle transport container. To do so, the resource provider uses the `nk_get_badge_op()` function, which extracts the pointer to the resource transfer context from the handle transport container if the received permissions mask has the corresponding flags set for the requested operation.

e. The resource provider uses the resource transfer context and the user resource context to perform the corresponding operation with the resource as requested by the resource consumer. Then the resource provider sends the results of this operation to the resource consumer.

f. The `UseResource()` method call completes successfully. The resource consumer receives the results of the operation performed with the resource.

4. The resource consumer calls the `CloseResource()` method to close access to the resource.

   a. The handle that was received from the resource provider at step 2 is used as a parameter of the `CloseResource()` method. Before calling this method, the resource consumer uses the `nk_handle_desc()` macro to package the handle into the handle transport container. After the `CloseResource()` method is called, the resource consumer uses the `KnHandleClose()` function to close the handle.

   b. The handle is dereferenced, after which the resource provider receives the pointer to the resource transfer context.

   c. The resource provider uses the `nk_is_handle_dereferenced()` function to verify that the dereferencing operation was completed instead of a handle transfer.

   d. The resource provider uses the `nk_get_badge()` function to extract the pointer to the resource transfer context from the handle transport container.

   e. The resource provider uses the `KnHandleRevokeSubtree()` function to revoke the handle owned by the resource consumer. The resource handle owned by the resource provider and the handle of the resource transfer context object are used as parameters of this function. The resource provider obtains access to these handles through the pointer to the resource transfer context. (Technically, the handle owned by the resource consumer does not have to be revoked because the resource consumer already closed it. However, the revoke operation is performed in case the resource provider is not sure if the resource consumer actually closed the handle).

   f. The `CloseResource()` method call completes successfully.

5. The resource provider frees up the memory that was allocated for the resource transfer context and user resource context.

   a. The resource provider calls the `KnNoticeGetEvent()` function to receive a notification that the resource transfer context object was closed, and uses the `KnHandleClose()` function to close the handle of the resource transfer context object.

   b. The resource provider calls the `KnNoticeGetEvent()` function to receive a notification that the resource transfer context object was deleted, and frees up the memory that was allocated for the resource transfer context.

   c. The resource provider uses the `KnHandleClose()` function to close the resource handle and free up the memory that was allocated for the user resource context.

Managing virtual memory (vmm_api.h)

The API is defined in the header file `sysroot-*-kos/include/coresrv/vmm/vmm_api.h` from the KasperskyOS SDK.

The API is designed to allocate and free memory, create shared memory, and prepare ELF image segments to be loaded into process memory.

Allocating and freeing memory

Information about API functions is provided in the table below.

Using the API

Virtual memory pages may be free, reserved, or committed. Free pages that are used to allocate a virtual memory region become reserved pages. Reserved pages may or may not be mapped to physical memory. Reserved pages that are mapped to physical memory are committed pages.

Pages of a virtual memory region that are allocated by the `KnVmAllocate()` function call can be committed in three different ways:

1. Fully committed when the region is allocated.
2. Fully or partially committed after allocation of the region (by calling the `KnVmCommit()` function).
3. Committed whenever virtual addresses are accessed (so-called "lazy" mode).

When the first option is used, the entire required volume of physical memory is allocated immediately after the virtual memory region is reserved. When the second or third option is used, the virtual memory region is reserved without allocating physical memory. This lets you conserve physical memory if a reserved virtual memory region will not actually be required or will only be partially used. In addition, a virtual memory region can be allocated faster if its allocation does not include commitment.

When in "lazy" mode, physical memory is allocated only when the virtual memory region is actually queried. In this case, the page containing the accessed address and several pages before and after this address are committed.

If you call the `KnVmAllocate()` function with the `VMM_FLAG_COMMIT` and `VMM_FLAG_LOCKED` flags, the virtual memory region will be reserved and committed. If you call the `KnVmAllocate()` function with the `VMM_FLAG_LOCKED` flag but without the `VMM_FLAG_COMMIT` flag, a virtual memory region will be reserved and you will need to use the `KnVmCommit()` function to commit this region. If you call the `KnVmAllocate()` function with the `VMM_FLAG_COMMIT` flag but without the `VMM_FLAG_LOCKED` flag, the virtual memory region will be reserved and committed only in "lazy" mode. If you call the `KnVmAllocate()` function without the `VMM_FLAG_COMMIT` and `VMM_FLAG_LOCKED` flags, a virtual memory region will be reserved and this region will be committed in lazy mode, but only after the `KnVmCommit()` function is applied. Committing virtual memory in lazy mode, which is permitted by default in the system, can be prohibited at the system level by defining the kernel startup parameter `vmm.overcommit=disabled`, or in a separate process by setting the `KOS_TASK_FLAG_LOCK_MEM` flag when creating a process using the task.h API. In this case, the `KnVmAllocate()` and `KnVmCommit()` functions either commit virtual memory and allocate the entire amount of necessary physical memory, or end with an error if there is insufficient physical memory, even if the `flags` parameter of the `KnVmAllocate()` function requires committing in lazy mode.

By default, the system is configured so that a thread querying virtual memory that must be committed in lazy mode is locked while waiting for memory to be freed if there is insufficient physical memory. If you define the kernel startup parameter `vmm.overcommit=exception`, an exception of the `EXCEPTION_OVERCOMMIT` type is thrown instead of locking the thread. (The `EXCEPTION_OVERCOMMIT` constant is defined in the header file `sysroot-*-kos/include/hal/exc_codes.h` from the KasperskyOS SDK.) In a separate process, you can override the system-wide setting. To do so, you need to specify the `KOS_TASK_FLAG_OVERCOMMIT_EXCEPTION` or `KOS_TASK_FLAG_OVERCOMMIT_WAIT_FREE` flag when creating a process using the task.h API.

A virtual memory region is allocated when the MDL buffer, DMA buffer or MMIO memory region is mapped to process memory. This region is allocated by the mapping function.

A guard page may reside at the beginning and/or end of the virtual memory region. This page is never committed. Any query of this page will result in an exception signaling that the region boundaries have been exceeded.

To change the access rights to the virtual memory region, call the `KnVmProtect()` function. You can fully close and then re-open access to a region while retaining is contents.

To free physical memory while retaining the reservation of virtual addresses, call the `KnVmDecommit()` or `KnVmReset()` function. In this case, the contents of the virtual memory region will be lost. After freeing physical memory by calling the `KnVmDecommit()` function, you must call the `KnVmCommit()` function to subsequently use the virtual memory region. After freeing physical memory by calling the `KnVmReset()` function, the virtual memory region can be used without any additional actions required. This virtual memory region will correspond to memory allocated by calling the `KnVmAllocate()` function with the `VMM_FLAG_COMMIT` flag but without the `VMM_FLAG_LOCKED` flag.

> The `KnVmProtect()`, `KnVmDecommit()`, and `KnVmReset()` functions cannot be used if the MDL buffer, DMA buffer or MMIO memory region is mapped to the virtual memory region.

To free the virtual memory region, call the `KnVmUnmap()` function. When this function is called, the reserved pages will become free regardless of whether or not they were committed, and the physical memory mapped to committed pages is also freed.

> The `KnVmUnmap()` function frees the virtual addresses of a region mapped to the MDL buffer, but does not flush the MDL buffer. In addition, this function cannot be used if the DMA buffer or MMIO memory region is mapped to the virtual memory region.

The `KnVmCommit()`, `KnVmProtect()`, `KnVmDecommit()`, `KnVmReset()`, and `KnVmUnmap()` functions can be applied for the entire allocated virtual memory region or for a portion of it.

The functions presented in this section provide the basis for implementing memory allocation/deallocation functions of the libkos library, and functions of POSIX interfaces such as `malloc()`, `calloc()`, `realloc()`, `free()`, `mmap()`, and `munmap()`.

Information about API functions

vmm_api.h functions

Function	Information about the function
`KnVmAllocate()`	Purpose Allocates (reserves and optionally commits) a virtual memory region. Parameters • [in,optional] `addr` – page-aligned, preferred base address of the virtual memory region, or `RTL_NULL` to select this address automatically. • [in] `size` – size (in bytes) of the virtual memory region. It must be a multiple of the memory page size. • [in] `flags` – flags defining the parameters of the virtual memory region. The flags are defined in the header file `sysroot-*-kos/include/vmm/flags.h` from the KasperskyOS SDK. Returned values If successful, the function returns the base address of the allocated virtual memory region, otherwise it returns `RTL_NULL`. Additional information In the `flags` parameter, you can specify the following flags: • `VMM_FLAG_COMMIT`, `VMM_FLAG_LOCKED` – flags that define how the virtual memory region is committed. • `VMM_FLAG_READ`, `VMM_FLAG_WRITE`, `VMM_FLAG_EXECUTE` and `VMM_FLAG_RWX_MASK` – flags defining the access rights to the virtual memory region. • `VMM_FLAG_LOW_GUARD`, `VMM_FLAG_HIGH_GUARD` – adds a guard page to the beginning and end of the virtual memory region, respectively. The size of the guard page is not included in the size of the virtual memory region. • `VMM_FLAG_GROW_DOWN` – defines the direction in which the virtual memory is mapped to physical memory when "lazy" commitment mode is applied. If this flag is set, the virtual memory page containing the accessed address and several preceding pages are mapped to physical memory. If this flag is not set, the virtual memory page containing the accessed address and several subsequent pages are mapped to physical memory. Permissible combinations of flags defining the access rights to the virtual memory region: • `VMM_FLAG_READ` – read access. • `VMM_FLAG_READ | VMM_FLAG_WRITE` – read-and-write access. • `VMM_FLAG_READ | VMM_FLAG_EXECUTE` – read-and-execute access. • `VMM_FLAG_RWX_MASK` or `VMM_FLAG_READ | VMM_FLAG_WRITE | VMM_FLAG_EXECUTE` – read/write/execute access. • `0` – access denied.
`KnVmCommit()`	Purpose Commits a virtual memory region. Parameters • [in] `addr` – page-aligned base address of the virtual memory region. • [in] `size` – size (in bytes) of the virtual memory region. It must be a multiple of the memory page size. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KnVmDecommit()`	Purpose Decommits a virtual memory region. Parameters • [in] `addr` – page-aligned base address of the virtual memory region. • [in] `size` – size (in bytes) of the virtual memory region. It must be a multiple of the memory page size. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnVmProtect()`	Purpose Modifies the access rights to the virtual memory region. Parameters • [in] `addr` – page-aligned base address of the virtual memory region. • [in] `size` – size (in bytes) of the virtual memory region. It must be a multiple of the memory page size. • [in] `newFlags` – flags defining the access rights to the virtual memory region. The flags are defined in the header file `sysroot-*-kos/include/vmm/flags.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information In the `newFlags` parameter, you can specify the following combinations of flags: • `VMM_FLAG_READ` – read access. • `VMM_FLAG_READ | VMM_FLAG_WRITE` – read-and-write access. • `VMM_FLAG_READ | VMM_FLAG_EXECUTE` – read-and-execute access. • `VMM_FLAG_RWX_MASK` or `VMM_FLAG_READ | VMM_FLAG_WRITE | VMM_FLAG_EXECUTE` – read/write/execute access. • `0` – access denied.
`KnVmUnmap()`	Purpose Frees up the virtual memory region. Parameters • [in] `addr` – page-aligned base address of the virtual memory region. • [in] `size` – size (in bytes) of the virtual memory region. It must be a multiple of the memory page size. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KnVmReset()`	Purpose Decommits a virtual memory region. Parameters • [in] `addr` – page-aligned base address of the virtual memory region. • [in] `size` – size (in bytes) of the virtual memory region. It must be a multiple of the memory page size. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Creating shared memory

Information about API functions is provided in the table below.

Using the API

An MDL buffer is used to create shared memory. An *MDL buffer* consists of one or more physical memory regions that can be mapped to the memory of multiple processes at the same time. A kernel object known as a memory descriptor list is used to map the MDL buffer to process memory. A *memory descriptor list* (MDL) is a data structure containing the addresses and sizes of physical memory regions comprising the MDL buffer. The MDL buffer handle identifies the memory descriptor list.

To create shared memory, a process needs to create an MDL buffer, map it to its own memory, and pass the MDL buffer handle via IPC to other processes that also need to map this MDL buffer to their own memory.

To create an MDL buffer, call the `KnPmmMdlCreate()`, `KnPmmMdlCreateFromBuf()`, `KnPmmMdlCreateFromVm()` or `KnPmmMdlClone()` function. The `KnPmmMdlCreateFromBuf()` function creates an MDL buffer and copies data to it. The `KnPmmMdlCreateFromVm()` function creates an MDL buffer and maps it to the memory of the calling process. The `KnPmmMdlClone()` function creates an MDL buffer based on an existing one.

To reserve a virtual memory region and map the MDL buffer to it, call the `KnPmmMdlMap()` function. An MDL buffer can be mapped to multiple virtual memory regions of the same process.

An MDL buffer can be used to transfer large volumes of data between processes without creating shared memory. In this case, you must make sure that the MDL buffer is not mapped to the memory of multiple processes at the same time. Interacting processes must take turns mapping the MDL buffer to their own memory, reading and/or writing data and then freeing the virtual memory region that is mapped to this MDL buffer.

> The Kaspersky Security Module cannot control data that is transferred between processes via the MDL buffer.

Flushing an MDL buffer

To flush an MDL buffer, complete the following steps:

1. Free the virtual memory regions mapped to the MDL buffer in all processes that are using this MDL buffer.

   To complete this step, use the `KnVmUnmap()` function.

2. Close or revoke each MDL buffer handle in all processes that own these handles.

   To complete this step, use the `KnHandleClose()` and/or `KnHandleRevoke()` functions that are declared in the header file `sysroot-*-kos/include/coresrv/handle/handle_api.h` from the KasperskyOS SDK.

Information about API functions

vmm_api.h functions

Function	Information about the function
`KnPmmMdlCreate()`	Purpose Creates an MDL buffer. Parameters • [in] `size` – size (in bytes) of the MDL buffer. It must be a multiple of the memory page size. • [in] `flags` – flags defining the access rights to the MDL buffer. The parameter type and flags are defined in the header file `sysroot-*-kos/include/vmm/flags.h` from the KasperskyOS SDK. • [out] `outHandle` – pointer to the MDL buffer handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information In the `flags` parameter, you can specify the following combinations of flags: • `VMM_FLAG_READ` – read access. • `VMM_FLAG_READ | VMM_FLAG_WRITE` – read-and-write access. • `VMM_FLAG_READ | VMM_FLAG_EXECUTE` – read-and-execute access. • `VMM_FLAG_RWX_MASK` or `VMM_FLAG_READ | VMM_FLAG_WRITE | VMM_FLAG_EXECUTE` – read/write/execute access.

Function	Information about the function
`KnPmmMdlCreateFromVm()`	Purpose Creates an MDL buffer from physical memory that is mapped to the defined virtual memory region and maps the created MDL buffer to this region. Parameters • [in] `addr` – page-aligned base address of the virtual memory region. • [in] `size` – size (in bytes) of the virtual memory region. It must be a multiple of the memory page size. • [in] `flags` – flags defining the access rights to the MDL buffer. The parameter type and flags are defined in the header file `sysroot-*-kos/include/vmm/flags.h` from the KasperskyOS SDK. • [out] `outHandle` – pointer to the MDL buffer handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information This function can be used if the defined virtual memory region is allocated with commitment. In the `flags` parameter, you can specify the following combinations of flags: • `VMM_FLAG_READ` – read access. • `VMM_FLAG_READ | VMM_FLAG_WRITE` – read-and-write access. • `VMM_FLAG_READ | VMM_FLAG_EXECUTE` – read-and-execute access. • `VMM_FLAG_RWX_MASK` or `VMM_FLAG_READ | VMM_FLAG_WRITE | VMM_FLAG_EXECUTE` – read/write/execute access.

Function	Information about the function
`KnPmmMdlCreateFromBuf()`	Purpose Creates an MDL buffer and copies data from the defined buffer to the MDL buffer. Parameters • [in,optional] `offset` – offset (in bytes) in the MDL buffer where the data write operation must start. This offset is ignored if the `bufSize` parameter is set to `0`. • [in] `size` – size (in bytes) of the MDL buffer. It must be a multiple of the memory page size. Also, if the `bufSize` parameter is set to a value other than zero, the following condition must be met: *size>=bufSize+offset*. • [in] `flags` – flags defining the access rights to the MDL buffer. The parameter type and flags are defined in the header file `sysroot-*-kos/include/vmm/flags.h` from the KasperskyOS SDK. • [in,optional] `buf` – pointer to the buffer containing the data. You can specify `RTL_NULL` if the `bufSize` parameter is set to `0`. • [in] `bufSize` – size of the buffer containing the data (in bytes), or `0` to create an MDL buffer without copying data to it (equivalent to a `KnPmmMdlCreate()` function call). • [out] `outHandle` – pointer to the MDL buffer handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information In the `flags` parameter, you can specify the following combinations of flags: • `VMM_FLAG_READ` – read access. • `VMM_FLAG_READ | VMM_FLAG_WRITE` – read-and-write access. • `VMM_FLAG_READ | VMM_FLAG_EXECUTE` – read-and-execute access. • `VMM_FLAG_RWX_MASK` or `VMM_FLAG_READ | VMM_FLAG_WRITE | VMM_FLAG_EXECUTE` – read/write/execute access.

Function	Information about the function
`KnPmmMdlMap()`	Purpose Reserves a virtual memory region and maps the MDL buffer to it. Parameters • [in] `handle` – MDL buffer handle. • [in] `offset` – offset (in bytes) in the MDL buffer where mapping should start. It must be a multiple of the memory page size. • [in] `length` – size (in bytes) of the part of the MDL buffer that needs to be mapped. It must be a multiple of the memory page size. The following condition must also be fulfilled: *length<=size of MDL buffer-offset*. • [in,optional] `hint` – page-aligned, preferred base address of the virtual memory region, or `RTL_NULL` to select this address automatically. • [in] `prots` – flags defining the parameters of the virtual memory region. The flags are defined in the header file `sysroot-*-kos/include/vmm/flags.h` from the KasperskyOS SDK. • [out] `addr` – pointer to the base address of the virtual memory region. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information In the `prots` parameter, you can specify the following flags: • `VMM_FLAG_READ`, `VMM_FLAG_WRITE`, `VMM_FLAG_EXECUTE` and `VMM_FLAG_RWX_MASK` – flags defining the access rights to the virtual memory region. • `VMM_FLAG_LOW_GUARD`, `VMM_FLAG_HIGH_GUARD` – adds a protected page to the beginning and end of the virtual memory region, respectively. Permissible combinations of flags defining the access rights to the virtual memory region: • `VMM_FLAG_READ` – read access. • `VMM_FLAG_READ | VMM_FLAG_WRITE` – read-and-write access. • `VMM_FLAG_READ | VMM_FLAG_EXECUTE` – read-and-execute access. • `VMM_FLAG_RWX_MASK` or `VMM_FLAG_READ | VMM_FLAG_WRITE | VMM_FLAG_EXECUTE` – read/write/execute access.
`KnPmmMdlClone()`	Purpose Creates an MDL buffer based on an existing one. The MDL buffer is created from the same regions of physical memory as the original buffer. Parameters • [in] `origin` – handle of the original MDL buffer. • [in] `offset` – offset (in bytes) in the original MDL buffer where duplication should start. It must be a multiple of the memory page size. • [in] `length` – size (in bytes) of the part of the original MDL buffer that needs to be duplicated. It must be a multiple of the memory page size. The following condition must also be fulfilled: *length<=size of original MDL buffer-offset*. • [out] `clone` – pointer to the handle of the created MDL buffer. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Preparing ELF image segments to be loaded into process memory

Information about API functions is provided in the table below.

Using the API

MDL buffers are used not only to create shared memory, but also to load ELF image segments into the memory of a new process. (ELF image segments are loaded by the `Einit` initializing program, for example.)

The `KnVmSegInitFromVm()` and `KnVmSegInitFromBuf()` functions create an MDL buffer and put the ELF image segment into it so that this segment can then be loaded into the memory of a new process.

Flushing MDL buffers containing ELF image segments

To flush MDL buffers containing ELF image segments, you must terminate the new process whose memory is mapped to these MDL buffers. You also need to complete the following steps before or after termination of this process:

1. Free the virtual memory regions that are mapped to the MDL buffers in the process that created these MDL buffers.

   This step is required only for MDL buffers that were created using the `KnVmSegInitFromVm()` function.

   To complete this step, use the `KnVmUnmap()` function.

2. Close the handles of MDL buffers in the process that created these MDL buffers.

   To complete this step, use the `KnHandleClose()` function, which is declared in the header file `sysroot-*-kos/include/coresrv/handle/handle_api.h` from the KasperskyOS SDK.

Information about API functions

vmm_api.h functions

Function	Information about the function
`KnVmSegInitFromVm()`	Purpose Creates an MDL buffer from physical memory that is mapped to the specified virtual memory region containing the ELF image segment. Parameters • [out] `seg` – pointer to the structure describing the ELF image segment. This structure contains the handle of the MDL buffer (the `h` field) and is used to load an ELF image segment into process memory. The type of structure is defined in the header file `sysroot-*-kos/include/coresrv/vmm/vmm_types.h` from the KasperskyOS SDK. • [in,optional] `loadAddr` – page-aligned address for loading the ELF image segment into process memory, or `0` to select this address automatically. • [in] `addr` – page-aligned base address of the virtual memory region. • [in] `size` – size (in bytes) of the virtual memory region. It must be a multiple of the memory page size. • [in] `flags` – flags defining the access rights to the MDL buffer. The parameter type and flags are defined in the header file `sysroot-*-kos/include/vmm/flags.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information This function can be used if the defined virtual memory region is allocated with commitment. In the `flags` parameter, you can specify the following combinations of flags: • `VMM_FLAG_READ` – read access. • `VMM_FLAG_READ | VMM_FLAG_WRITE` – read-and-write access. • `VMM_FLAG_READ | VMM_FLAG_EXECUTE` – read-and-execute access. • `VMM_FLAG_RWX_MASK` or `VMM_FLAG_READ | VMM_FLAG_WRITE | VMM_FLAG_EXECUTE` – read/write/execute access.

Function	Information about the function
`KnVmSegInitFromBuf()`	Purpose Creates an MDL buffer and copies the ELF image segment from the defined buffer to the MDL buffer. Parameters • [out] `seg` – pointer to the structure describing the ELF image segment. This structure contains the handle of the MDL buffer (the `h` field) and is used to load an ELF image segment into process memory. The type of structure is defined in the header file `sysroot-*-kos/include/coresrv/vmm/vmm_types.h` from the KasperskyOS SDK. • [in,optional] `loadAddr` – page-aligned address for loading the ELF image segment into process memory, or `0` to select this address automatically. • [in] `offset` – offset in the MDL buffer where writing of the ELF image segment must start (in bytes). • [in] `size` – size (in bytes) of the ELF image segment. It must be a multiple of the memory page size. • [in] `flags` – flags defining the access rights to the MDL buffer. The parameter type and flags are defined in the header file `sysroot-*-kos/include/vmm/flags.h` from the KasperskyOS SDK. • [in] `fileOffset` – offset (in bytes) of the segment in the ELF image. • [in] `buildId` – pointer to the build ID. The linker writes this ID to the ELF image. • [in] `buildIdSize` – size (in bytes) of the build ID. • [in] `buf` – pointer to the buffer containing the ELF image segment. • [in] `bufSize` – size (in bytes) of the buffer containing the ELF image segment. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information In the `flags` parameter, you can specify the following combinations of flags: • `VMM_FLAG_READ` – read access. • `VMM_FLAG_READ | VMM_FLAG_WRITE` – read-and-write access. • `VMM_FLAG_READ | VMM_FLAG_EXECUTE` – read-and-execute access. • `VMM_FLAG_RWX_MASK` or `VMM_FLAG_READ | VMM_FLAG_WRITE | VMM_FLAG_EXECUTE` – read/write/execute access.

Receiving information about memory

Information about API functions is provided in the table below.

Getting information about a virtual memory page

To get information about the virtual memory page, call the `KnVmQuery()` function.

Get the MDL buffer size

To get the MDL buffer size, call the `KnPmmMdlGetSize()` function.

Getting information about a virtual memory region that is mapped to an MDL buffer

MDL buffers are used not only to create shared memory and to load ELF image segments into the memory of a new process, but also to create a memory dump from a process that is in a "frozen" state. The `KnTaskGetNextVmRegion()` function from the task_api.h API creates an MDL buffer from physical memory that is mapped to a virtual memory region of a process that is in a frozen state. The memory descriptor list from this MDL buffer may contain fictitious physical memory regions for which a size is defined but no address is specified. Fictitious physical memory regions in the memory descriptor list of the MDL buffer correspond to sections of non-committed pages in a virtual memory region of a process that is in a frozen state. The MDL buffer that was created by the `KnTaskGetNextVmRegion()` function must be mapped to the virtual memory region of the process that created the dump to then save its contents. This mapping is done by calling the `KnPmmMdlMap()` function. This virtual memory region will contain sections of non-committed pages if the memory descriptor list of the MDL buffer contains fictitious physical memory regions. If the process that created the dump will query sections of non-committed pages in the virtual memory region that is mapped to the MDL buffer, this will lead to an undesirable allocation of physical memory. Information about the arrangement of committed and non-committed pages in a virtual memory region that is mapped to the MDL buffer will allow the process that created the dump to avoid querying sections of non-committed pages in this virtual memory region.

To receive information about the arrangement of committed and non-committed pages in a virtual memory region that is mapped to the MDL buffer, you must use the `KnPmmMdlGetUsageMap()` function. The function uses the `counters` parameter to pass the array of counters of committed and non-committed pages. The counters of committed and non-committed pages are strictly alternated. This indicates that a section of committed pages is followed by a section of non-committed pages, and vice versa. The output parameter `firstUsed` lets you define which counter is the first in the array: the counter of committed pages or the counter of non-committed pages.

To collect information about one virtual memory region, you may need to call the `KnPmmMdlGetUsageMap()` function again for the following reasons:

- The maximum number of counters passed by the function during one call is limited.
- The function stops collecting information if the counter of committed or non-committed pages reaches the maximum value.

When the `KnPmmMdlGetUsageMap()` function is called again, you must use the `startPage` parameter to specify the offset, which is the number of pages whose information was collected during previous calls of the function.

To make sure that information was collected for the entire virtual memory region that is mapped to the MDL buffer, you must verify that one of the following conditions is fulfilled:

- The sum of the values of counters received as a result of all executed calls of the `KnPmmMdlGetUsageMap()` function matches the size of the MDL buffer.
- The actual number of counters passed by the `KnPmmMdlGetUsageMap()` function via the `usedCounters` parameter when the call is repeated is equal to zero.

Information about API functions

vmm_api.h functions

Function	Information about the function
`KnVmQuery()`	Purpose Gets information about a virtual memory page. Parameters • [in] `addr` – address included in the virtual memory page. • [out] `info` – pointer to the structure containing information about the virtual memory page. The type of structure is defined in the header file `sysroot-*-kos/include/vmm/info.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnPmmMdlGetSize()`	Purpose Gets the size of the MDL buffer. Parameters • [in] `handle` – MDL buffer handle. • [out] `size` – size (in bytes) of the MDL buffer. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnPmmMdlGetUsageMap()`	Purpose Receives information about the arrangement of committed and non-committed pages in a virtual memory region that is mapped to the MDL buffer. Parameters • [in] `handle` – MDL buffer handle. • [in] `startPage` – virtual memory region offset from which the information collection starts in memory pages. • [out] `firstUsed` – pointer to the boolean value indicating which counter is the first counter in the `counters` parameter: committed pages counter (`rtl_true`) or non-committed pages counter (`rtl_false`). • [in,out] `usedCounters` – input data: pointer to the maximum number of counters in the array defined via the `counters` parameter (must not exceed the value of the `MappingUsesSlots` constant defined in the file named `sysroot-*-kos/include/kl/core/VMM.idl` from the KasperskyOS SDK); output data: pointer to the actual number of counters in the array defined via the `counters` parameter. • [out] `counters` – pointer to the array containing counters of committed and non-committed pages. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Using a working set (ws_api.h)

The API is defined in the header file `sysroot-*-kos/include/coresrv/vmm/ws_api.h` from the KasperskyOS SDK.

The API is intended for performing operations with a working set. *Working set* (WS) is an abstraction that describes the use of physical memory in a system. This is a KasperskyOS kernel object associated with data such as the total number of physical memory pages in the system, the number of occupied physical memory pages in the system, and threshold values for the amount of occupied physical memory in the system.

The working set contains two threshold values (upper and lower) for the amount of occupied physical memory in the system. The kernel tracks the number of occupied pages of physical memory and emits a signal when it reaches the defined thresholds. If the current number of occupied pages is below the upper threshold, attainment of the upper threshold is monitored. If the current number of occupied pages is not below the upper threshold, attainment of the lower threshold is monitored. When the upper threshold is reached, the kernel starts tracking attainment of the lower threshold, and vice versa.

If the number of occupied pages of physical memory in the system reaches a threshold value, the kernel associates the event with the working set by specifying the EVENT_WS_THRESHOLD flag in the `event mask`. To receive notifications about events related to the working set, use the notice_api.h API.

The API functions are thread-safe.

Information about API functions is provided in the table below.

Opening access to a working set

To open access to a working set, call the `KnPmmWsOpen()` function. Access to a working set can be opened multiple times in one or more processes. The KnPmmWsOpen() function uses the outHandle output parameter to pass the handle of the working set. This handle is an input parameter not only for all other functions of this API, but also for the notice_api.h API functions that are used to configure the notification receiver.

If access to the working set is no longer required, its handle must be closed.

Getting the characteristics of a working set

To get the characteristics of a working set, call the `KnPmmWsGetInfo()` function.

A working set has the following characteristics:

- Total number of pages of physical memory in the system (`maxSize`)
- Number of occupied pages of physical memory in the system (`currentSize`)
- Lower threshold value for the amount of occupied physical memory in the system (`lowThreshold`)
- Upper threshold value for the amount of occupied physical memory in the system (`highThreshold`)
- Characteristic that indicates whether threshold values are set for the amount of occupied physical memory in the system (`nextThresholdToBeTriggered`)

The `nextThresholdToBeTriggered` characteristic can have the following values:

- `WsNoThresholdIsActive` – no threshold values have been set.
- `WsThresholdHighIsNext` – the upper threshold value is being monitored.
- `WsThresholdLowIsNext` – the lower threshold value is being monitored.

Setting and removing threshold values for the amount of occupied physical memory in the system

To set the threshold values for the amount of occupied physical memory in the system, call the `KnPmmWsSetThreshold()` function. Prior to the first call to this function, the threshold values are zero. As a result of this call, the kernel begins to track the attainment of the set threshold values. Threshold values cannot be set until the previously set values are removed.

To remove the threshold values for the amount of occupied physical memory in the system, call the `KnPmmWsClearThreshold()` function. As a result of this call, the kernel stops tracking the attainment of the previously set threshold values and these threshold values are cleared.

Threshold values for the amount of occupied physical memory in the system may be removed and re-set by another process or by different code in the same process.

Information about API functions

ws_api.h functions

Function	Information about the function
`KnPmmWsOpen()`	Purpose Provides access to the working set. Parameters • [out] `outHandle` – pointer to the working set handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnPmmWsGetInfo()`	Purpose Gets the characteristics of the working set. Parameters • [in] `ws` – working set handle. • [out] `info` – pointer to the structure for storing the characteristics of the working set. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KnPmmWsSetThreshold()`	Purpose Sets the threshold values for the amount of used physical memory in the system. Parameters • [in] `ws` – working set handle. • [in] `threshold` – the lower threshold value, in memory pages. • [in] `width` – the offset of the upper threshold value relative to the lower one, in memory pages. The sum of `threshold` and `width` gives the upper threshold value. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If threshold values have already been set, the function returns `rcAlreadyDone`.
`KnPmmWsClearThreshold()`	Purpose Delete the threshold values for the amount of used physical memory in the system. Parameters • [in] `ws` – working set handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Allocating and freeing memory (alloc.h)

The API is defined in the header file `sysroot-*-kos/include/kos/alloc.h` from the KasperskyOS SDK.

The API is intended for allocating and freeing memory. Allocated memory is a committed virtual memory region that can be accessed for read-and-write operations.

Information about API functions is provided in the table below.

alloc.h functions

Function	Information about the function
`KosMemAllocEx()`	Purpose Allocates memory. Parameters • [in,optional] `size` – size (in bytes) of the allocated memory. If you specify a size of zero, the function returns a pointer that can be passed to the `KosMemFree()` function. • [in] `align` – value defining the alignment of the allocated memory. It must be a power of two. The address of allocated memory can be unaligned (*align=1*) or aligned (*align=2,4,...,2^N*) to the boundary of a *2^N*-byte sequence (for example, two-byte or four-byte). • [in] `zeroed` – value defining the initialization of the allocated memory (`1` – initialize with zeros, `0` – do not initialize). Returned values If successful, the function returns the pointer to the allocated memory, otherwise it returns `RTL_NULL`.

Function	Information about the function
`KosMemAlloc()`	Purpose Allocates memory. Parameters • [in,optional] `size` – size (in bytes) of the allocated memory. If you specify a size of zero, the function returns a pointer that can be passed to the `KosMemFree()` function. Returned values If successful, the function returns the pointer to the allocated memory, otherwise it returns `RTL_NULL`.
`KosMemZalloc()`	Purpose Allocates memory and initializes it with zeros. Parameters • [in,optional] `size` – size (in bytes) of the allocated memory. If you specify a size of zero, the function returns a pointer that can be passed to the `KosMemFree()` function. Returned values If successful, the function returns the pointer to the allocated memory, otherwise it returns `RTL_NULL`.
`KosMemFree()`	Purpose Deallocates memory. Parameters • [in,optional] `ptr` – pointer to the freed memory. If you specify `RTL_NULL`, the function does not perform any actions. Returned values N/A
`KosMemRealloc()`	Purpose Allocates memory and (optionally) copies the contents of previously allocated memory to the newly allocated memory and frees the previously allocated memory after the copy operation is complete. Parameters • [in,optional] `ptr` – pointer to the previously allocated memory, or `RTL_NULL` if you only need to allocate memory without copying the contents of previously allocated memory. • [in,optional] `size` – size (in bytes) of the newly allocated memory. The contents of previously allocated memory are copied to the newly allocated memory in whatever amount that can be accommodated in the newly allocated memory. If you specify a size of zero, the function returns a pointer that can be passed to the `KosMemFree()` function. Returned values If successful, the function returns the pointer to the allocated memory, otherwise it returns `RTL_NULL`.

Function	Information about the function
`KosMemReallocEx()`	Purpose Allocates memory and (optionally) copies the contents of previously allocated memory to the newly allocated memory and frees the previously allocated memory after the copy operation is complete. Parameters • [in,optional] `ptr` – pointer to the previously allocated memory, or `RTL_NULL` if you only need to allocate memory without copying the contents of previously allocated memory. • [in,optional] `size` – size (in bytes) of the newly allocated memory. The contents of previously allocated memory are copied to the newly allocated memory in whatever amount that can be accommodated in the newly allocated memory. If you specify a size of zero, the function returns a pointer that can be passed to the `KosMemFree()` function. • [in] `zeroed` – value defining the initialization of the allocated memory (`1` – initialize with zeros, `0` – do not initialize). If the `ptr` parameter is used to pass `RTL_NULL` and the `zeroed` parameter is set to `1`, the allocated memory will be initialized with zeros. If the size of the newly allocated memory is less than or equal to the size of the previously allocated memory, the value of the `zeroed` parameter does not affect the contents of the newly allocated memory. If the size of the newly allocated memory is greater than the size of the previously allocated memory, and the `zeroed` parameter is set to `1`, the fragment of newly allocated memory that is not occupied by the contents copied from the previously allocated memory may be initialized with zeros. To ensure that this fragment is initialized with zeros, you must do the following. The memory must be initially allocated by using either the `KosMemZalloc()` function, the `KosMemAllocEx()` function with the `zeroed` parameter set to `1`, or the `KosMemReallocEx()` function with the `zeroed` parameter set to `1` and the `ptr` parameter set to `RTL_NULL`. All subsequent `KosMemReallocEx()` calls must have the `zeroed` parameter set to `1`. Returned values If successful, the function returns the pointer to the allocated memory, otherwise it returns `RTL_NULL`.

Using DMA (dma.h)

The API is defined in the header file `sysroot-*-kos/include/coresrv/io/dma.h` from the KasperskyOS SDK.

The API is designed to set up data exchange between devices and RAM in *direct memory access* (DMA) mode in which the processor is not used.

Information about API functions is provided in the table below.

Using the API

The standard scenario for API usage includes the following steps:

1. Creating a DMA buffer.

   DMA buffer consists of one or more physical memory regions (blocks) that are used for DMA. A DMA buffer consisting of multiple blocks can be used if the device supports "scatter/gather DMA" mode. A DMA buffer consisting of one block can be used only if the device supports "scatter/gather DMA" or "continuous DMA" mode. The likelihood of creating a DMA buffer consisting of one large block is lower than the likelihood of creating a DMA buffer consisting of multiple small blocks. This is especially relevant when physical memory is highly fragmented.

   > If the device supports only "continuous DMA" mode, you must use a DMA buffer consisting of one block even if IOMMU is enabled.

To complete this step, call the `KnIoDmaCreate()`, `KnIoDmaAllocWithinRange()` or `KnIoDmaCreateContinuous()` function. The `KnIoDmaCreateContinuous()` function creates a DMA buffer consisting of one block. The `KnIoDmaCreate()` and `KnIoDmaAllocWithinRange()` functions create a DMA buffer consisting of one block if the *2^order* value is equal to the *memory page size* value, or if the *2^order* value is the next largest value of the *memory page size* in the ascending ordered set *{2^(order-1);memory page size;2^order}*. If the *memory page size* value is greater than the value of *2^order*, the `KnIoDmaCreate()` and `KnIoDmaAllocWithinRange()` functions can create a DMA buffer consisting of multiple blocks. In contrast to the `KnIoDmaCreate()` function, the `KnIoDmaAllocWithinRange()` function lets you specify the physical memory region in which the DMA buffer should be created.

The DMA buffer cache settings can be changed by calling the `KnIoDmaModify()` function.

The DMA buffer handle can be transferred to another process via IPC.

2. Mapping the DMA buffer to the memory of processes.

   One DMA buffer can be mapped to multiple virtual memory regions of one or more processes that own the handle of this DMA buffer. Mapping allows processes to receive read-and/or-write access to the DMA buffer.

   To reserve a virtual memory region and map the DMA buffer to it, call the `KnIoDmaMap()` function.

   A handle received when calling the `KnIoDmaMap()` function cannot be transferred to another process via IPC.

3. Opening access to the DMA buffer for a device via the `KnIoDmaBegin()` or `KnIoDmaBeginEx()` function call.

   The `KnIoDmaBegin()` or `KnIoDmaBeginEx()` function must be called to create a kernel object containing the addresses and sizes of blocks comprising the DMA buffer. A device needs this information to use the DMA buffer. A device can work with physical addresses and/or virtual addresses depending on whether IOMMU is enabled. If IOMMU is enabled, you must use the `KnIoDmaBegin()` or `KnIoDmaBeginEx()` function, and the object will contain virtual and physical addresses of blocks. If IOMMU is not enabled, you must use the `KnIoDmaBegin()` function, and the object will contain physical addresses of blocks only.

   When IOMMU is enabled, the device must be attached to an IOMMU domain.

   A handle received when calling the `KnIoDmaBegin()` or `KnIoDmaBeginEx()` function cannot be transferred to another process via IPC.

4. Information about the DMA buffer is received.

   At this step, get the addresses and sizes of blocks from the kernel object that was created by calling the `KnIoDmaBegin()` or `KnIoDmaBeginEx()` function. The received addresses and sizes will need to be passed to the device by using MMIO, for example. After receiving this information, the device can write to the DMA buffer and/or read from it.

   To complete this step, call one of the following functions:

   - `KnIoDmaGetInfo()`
   - `KnIoDmaGetPhysInfo()`
   - `KnIoDmaContinuousGetDmaAddr()`
   - `KnIoDmaContinuousGetPhysAddr()`

The `KnIoDmaGetInfo()` and `KnIoDmaGetPhysInfo()` functions get the memory page number (`frame`) and the `order` for each block. (The memory page number multiplied by the memory page size results in the block address. The *2^order* value is the block size in memory pages.) These functions use the output parameter `outInfo` to pass the `KosObject` that contains the structure with information about the DMA buffer (for details about `KosObject`s, see Using KosObjects (objects.h)). The type of structure is defined in the header file `sysroot-*-kos/include/io/io_dma.h` from the KasperskyOS SDK. The `KnIoDmaGetInfo()` function inserts data into all fields of the structure, while the `KnIoDmaGetPhysInfo()` function inserts data only into the `count` and `descriptors` fields while writing zeros to all other fields. When using the `KnIoDmaGetInfo()` function, the `descriptors` array contains virtual memory page numbers if IOMMU is enabled, and it contains physical memory page numbers if IOMMU is not enabled. When using the `KnIoDmaGetPhysInfo()` function, the `descriptors` array contains physical memory page numbers regardless of whether or not IOMMU is enabled.

The `KnIoDmaContinuousGetDmaAddr()` and `KnIoDmaContinuousGetPhysAddr()` functions can be used if the DMA buffer consists of one block. These functions let you get the block address. (The accepted block size should be the DMA buffer size that was defined when it was created.) The `KnIoDmaContinuousGetDmaAddr()` function uses the output parameter `addr` to pass the virtual address if IOMMU is enabled or the physical address if IOMMU is not enabled. The `KnIoDmaContinuousGetPhysAddr()` function uses the output parameter `addr` to pass the physical address, regardless of whether or not IOMMU is enabled.

The API includes the auxiliary functions named `KnIoDmaHandlesClear()` and `KnIoDmaHandlesClose()`. The `KnIoDmaHandlesClear()` function initializes the structure for storing the handles that are created at steps 1–3. The `KnIoDmaHandlesClose()` function closes the handles stored in this structure (by calling the `KnHandleClose()` function from the handle_api.h API) and initializes it. When the `KnIoDmaHandlesClose()` function is called, the absence of all or part of the handles in the structure does not lead to an error, and the available part of the handles will be closed. The `KnIoDmaHandlesClear()` and `KnIoDmaHandlesClose()` functions can be used as follows. Before performing step 1, initialize the structure for storing handles by calling the `KnIoDmaHandlesClear()` function. Then store the handles in this structure when performing steps 1–3. This will allow you to subsequently close all created handles with a single call of the `KnIoDmaHandlesClose()` function. For example, if an error occurs at one of the steps, you may need to close all already created handles to free the allocated resources.

Closing access to the DMA buffer for a device

Access to the DMA buffer can be closed for a device only if IOMMU is enabled. If you delete the kernel object that was created when the `KnIoDmaBegin()` or `KnIoDmaBeginEx()` function was called, device access to the DMA buffer will be closed. To delete this object, you must close its handle.

Flushing a DMA buffer

To flush a DMA buffer, complete the following steps:

1. Free the virtual memory regions that were reserved during `KnIoDmaMap()` function calls.

   To complete this step, you must close the handles that were received from `KnIoDmaMap()` function calls.

   This step must be completed for all processes whose memory is mapped to the DMA buffer.

2. Delete the kernel object that was created by the `KnIoDmaBegin()` or `KnIoDmaBeginEx()` function call.

   To complete this step, you must close the handle that was received when the `KnIoDmaBegin()` or `KnIoDmaBeginEx()` function was called.

3. Close or revoke each handle of the DMA buffer in all processes that own these handles.

You can use the `KnIoDmaHandlesClose()` function to close the handles when performing these steps.

Information about API functions

dma.h functions

Function	Information about the function
`KnIoDmaCreate()`	Purpose Creates a DMA buffer. Parameters • [in] `order` – parameter defining the minimum number of memory pages (*2^order*) in a block. • [in] `size` – size (in bytes) of the DMA buffer. It must be a multiple of the memory page size. • [in] `flags` – flags defining the DMA buffer parameters. The parameter type and flags are defined in the header file `sysroot-*-kos/include/io/io_dma.h` from the KasperskyOS SDK. • [out] `outRid` – pointer to the DMA buffer handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information In the `flags` parameter, you can specify the following flags: • `DMA_DIR_TO_DEVICE` – the device has read-access to the DMA buffer. • `DMA_DIR_FROM_DEVICE` – the device has write-access to the DMA buffer. • `DMA_DIR_BIDIR` – the device has read-and-write access to the DMA buffer. • `DMA_ZONE_DMA32` – only the first four gigabytes of physical memory can be used to create a DMA buffer. • `DMA_ATTR_WRITE_BACK`, `DMA_ATTR_WRITE_THROUGH`, `DMA_ATTR_CACHE_DISABLE`, `DMA_ATTR_WRITE_COMBINE`, `DMA_RULE_CACHE_VOLATILE`, `DMA_RULE_CACHE_FIXED` – cache management.

Function	Information about the function
`KnIoDmaCreateContinuous()`	Purpose Creates a DMA buffer consisting of one block. Parameters • [in] `size` – size (in bytes) of the DMA buffer. It must be a multiple of the memory page size. • [in] `flags` – flags defining the DMA buffer parameters. The parameter type and flags are defined in the header file `sysroot-*-kos/include/io/io_dma.h` from the KasperskyOS SDK. • [out] `outRid` – pointer to the DMA buffer handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information In the `flags` parameter, you can specify the following flags: • `DMA_DIR_TO_DEVICE` – the device has read-access to the DMA buffer. • `DMA_DIR_FROM_DEVICE` – the device has write-access to the DMA buffer. • `DMA_DIR_BIDIR` – the device has read-and-write access to the DMA buffer. • `DMA_ZONE_DMA32` – only the first four gigabytes of physical memory can be used to create a DMA buffer. • `DMA_ATTR_WRITE_BACK`, `DMA_ATTR_WRITE_THROUGH`, `DMA_ATTR_CACHE_DISABLE`, `DMA_ATTR_WRITE_COMBINE`, `DMA_RULE_CACHE_VOLATILE`, `DMA_RULE_CACHE_FIXED` – cache management.

Function	Information about the function
`KnIoDmaAllocWithinRange()`	Purpose Creates a DMA buffer within the specified region of physical memory. Parameters • [in] `from` – base address of the physical memory region. The parameter type is defined in the header file `sysroot-*-kos/include/hal/mtypes.h` from the KasperskyOS SDK. • [in] `to` – last address of the physical memory region. The parameter type is defined in the header file `sysroot-*-kos/include/hal/mtypes.h` from the KasperskyOS SDK. • [in] `order` – parameter defining the minimum number of memory pages (*2^order*) in a block. • [in] `size` – size (in bytes) of the DMA buffer. It must be a multiple of the memory page size. • [in] `flags` – flags defining the DMA buffer parameters. The parameter type and flags are defined in the header file `sysroot-*-kos/include/io/io_dma.h` from the KasperskyOS SDK. • [out] `outRid` – pointer to the DMA buffer handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information In the `flags` parameter, you can specify the following flags: • `DMA_DIR_TO_DEVICE` – the device has read-access to the DMA buffer. • `DMA_DIR_FROM_DEVICE` – the device has write-access to the DMA buffer. • `DMA_DIR_BIDIR` – the device has read-and-write access to the DMA buffer. • `DMA_ZONE_DMA32` – only the first four gigabytes of physical memory can be used to create a DMA buffer. • `DMA_ATTR_WRITE_BACK`, `DMA_ATTR_WRITE_THROUGH`, `DMA_ATTR_CACHE_DISABLE`, `DMA_ATTR_WRITE_COMBINE`, `DMA_RULE_CACHE_VOLATILE`, `DMA_RULE_CACHE_FIXED` – cache management.

Function	Information about the function
`KnIoDmaMap()`	Purpose Reserves a virtual memory region and maps the DMA buffer to it. Parameters • [in] `rid` – DMA buffer handle. • [in] `offset` – offset (in bytes) in the DMA buffer where mapping should start. It must be a multiple of the memory page size. • [in] `length` – size (in bytes) of the part of the DMA buffer that needs to be mapped. It must be a multiple of the memory page size. The following condition must also be fulfilled: *length<=size of DMA buffer-offset*. • [in,optional] `hint` – page-aligned, preferred base address of the virtual memory region, or `RTL_NULL` to select this address automatically. • [in] `vmflags` – flags defining the access rights to the virtual memory region. The flags are defined in the header file `sysroot-*-kos/include/vmm/flags.h` from the KasperskyOS SDK. • [out] `addr` – pointer to the base address of the virtual memory region. • [out] `handle` – pointer to the handle that is used to free the virtual memory region. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information In the `vmflags` parameter, you can specify the following flags: • `VMM_FLAG_READ` – read access. • `VMM_FLAG_WRITE` – write access.
`KnIoDmaModify()`	Purpose Modifies the DMA buffer cache settings. Parameters • [in] `rid` – DMA buffer handle. • [in] `newAttr` – flags defining the DMA buffer caching parameters. The flags are defined in the header file `sysroot-*-kos/include/io/io_dma.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information This function can be used if the following conditions are fulfilled: 1. The `DMA_RULE_CACHE_VOLATILE` flag was specified when the DMA buffer was created. 2. The DMA buffer is not mapped to virtual memory. 3. The `DMA_RULE_CACHE_VOLATILE` flag was specified during the previous function call (if completed). In the `newAttr` parameter, you can specify the following flags: • `DMA_ATTR_WRITE_BACK`, `DMA_ATTR_WRITE_THROUGH`, `DMA_ATTR_CACHE_DISABLE`, `DMA_ATTR_WRITE_COMBINE`, `DMA_RULE_CACHE_VOLATILE` – cache management.

Function	Information about the function
`KnIoDmaGetInfo()`	Purpose Gets information about a DMA buffer. This information includes the addresses and sizes of blocks. Parameters • [in] `rid` – DMA buffer handle. • [out] `outInfo` – pointer to the address of the object containing the structure with information about the DMA buffer. The type of structure is defined in the header file `sysroot-*-kos/include/io/io_dma.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information To delete an object that was received via the `outInfo` parameter, you must use the `KosPutObject()` function from the objects.h API.
`KnIoDmaGetPhysInfo()`	Purpose Gets the physical addresses and sizes of DMA buffer blocks. Parameters • [in] `rid` – DMA buffer handle. • [out] `outInfo` – pointer to the address of the object containing the structure with information about the DMA buffer. The type of structure is defined in the header file `sysroot-*-kos/include/io/io_dma.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information To delete an object that was received via the `outInfo` parameter, you must use the `KosPutObject()` function from the objects.h API.
`KnIoDmaContinuousGetDmaAddr()`	Purpose Gets the block address for a DMA buffer consisting of one block. Parameters • [in] `rid` – DMA buffer handle. • [out] `addr` – pointer to the block address. The type of address is defined in the header file `sysroot-*-kos/include/hal/mtypes.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KnIoDmaContinuousGetPhysAddr()`	Purpose Gets the block physical address for a DMA buffer consisting of one block. Parameters • [in] `rid` – DMA buffer handle. • [out] `addr` – pointer to the physical block address. The type of address is defined in the header file `sysroot-*-kos/include/hal/mtypes.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnIoDmaBegin()`	Purpose Opens access to a DMA buffer for a device. Parameters • [in] `rid` – DMA buffer handle. • [out] `handle` – pointer to the handle of the kernel object containing the addresses and sizes of blocks that were required for the device to use the DMA buffer. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnIoDmaBeginEx()`	Purpose Opens access to a DMA buffer for a device. Parameters • [in] `rid` – DMA buffer handle. • [in] `domain` – IOMMU domain handle. • [out] `handle` – pointer to the handle of the kernel object containing the addresses and sizes of blocks that were required for the device to use the DMA buffer. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnIoDmaHandlesClear()`	Purpose Initializes a structure for storing the handles of resources allocated for DMA provisioning. Parameters • [out] `handles` – pointer to the structure for storing handles. Returned values N/A Additional information After initialization, the structure fields take the `INVALID_HANDLE` value.

Function	Information about the function
`KnIoDmaHandlesClose()`	Purpose Closes the handles of resources allocated for DMA provisioning and initializes the structure in which these handles were stored. Parameters • [in, out] `handles` – pointer to a structure for storing handles. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information After initialization, the structure fields take the `INVALID_HANDLE` value.

Memory-mapped I/O (mmio.h)

The API is defined in the header file `sysroot-*-kos/include/coresrv/io/mmio.h` from the KasperskyOS SDK.

The API is intended for working with MMIO memory. *MMIO memory* consists of physical addresses that are mapped to registers and memory of devices. (Portions of one physical address space are used for physical memory addressing and access to registers and memory of devices.)

Information about API functions is provided in the table below.

Using the API

The standard scenario for API usage includes the following steps:

1. The MMIO memory region corresponding to the relevant device is registered.

   To complete this step, call the `KnRegisterPhyMem()` function. If the same MMIO memory region needs to be registered multiple times, the `MMIO_U_SHARED` flag must be specified in the `flags` parameter during each registration of this region. If the registered MMIO memory region is used by the kernel code, the `MMIO_K_SHARED` flag must be specified. If these flags are not specified, only the code that requests this access first (this can be either the kernel code or the code that is executed in user mode) will have access to the MMIO memory region. All subsequent registration attempts by calling the `KnRegisterPhyMem()` function will end with the `rcAlreadyExists` error, and messages regarding the problem with shared access to the MMIO memory region will be added to the diagnostic output. If the kernel fails to access the MMIO memory region because this region is already registered by the code being executed in user mode, a message regarding the problem with shared access to the MMIO memory region will also be added to the diagnostic output.

   The handle of an MMIO memory region can be transferred to another process via IPC.

2. Mapping the MMIO memory region to the memory of a process.

   An MMIO memory region can only be mapped to one virtual memory region of only one process at one time. (However, an MMIO memory region may be mapped to the virtual memory of different processes that own the handle of this region at different times.) Mapping allows the process to receive read-and/or-write access to the MMIO memory region.

   To reserve a virtual memory region and map the MMIO memory region to it, call the `KnIoMapMem()` function.

   A handle received when calling the `KnIoMapMem()` function cannot be transferred to another process via IPC.

3. Reading data from the MMIO memory region and writing data to it via process memory.

   The 8-, 16- and 32-bit words that are read from the MMIO memory region or written to it are values of device registers or contents of device memory.

   To complete this step, use the `IoReadMmReg8|16|32()` and `IoWriteMmReg8|16|32()` functions.

The API includes the auxiliary functions named `KnIoMemHandlesClear()` and `KnIoMemHandlesClose()`. The `KnIoMemHandlesClear()` function initializes the structure for storing the handles that are created at steps 1 and 2. The `KnIoMemHandlesClose()` function closes the handles stored in this structure (by calling the `KnHandleClose()` function from the handle_api.h API) and initializes it. When the `KnIoMemHandlesClose()` function is called, the absence of all or one handle in the structure does not result in an error, and the available handle will be closed. The `KnIoMemHandlesClear()` and `KnIoMemHandlesClose()` functions can be used as follows. Before performing step 1, initialize the structure for storing handles by calling the `KnIoMemHandlesClear()` function. Then store the handles in this structure when performing steps 1 and 2. This will allow you to subsequently close all created handles with a single call of the `KnIoMemHandlesClose()` function.

Deregistering an MMIO memory region

To deregister an MMIO memory region, complete the following steps:

1. Free the virtual memory region that was allocated when the `KnIoMapMem()` function was called.

   To perform this step, you must close the handle that was obtained when calling the `KnIoMapMem()` function.

2. Close or revoke each handle of the MMIO memory region in all processes that own those handles.

You can use the `KnIoMemHandlesClose()` function to close the handles when performing these steps.

Information about API functions

mmio.h functions

Function(s)	Information about the function(s)
`KnRegisterPhyMem()`	Purpose Registers an MMIO memory region. Parameters • [in] `addr` – page-aligned base address of the MMIO memory region. • [in] `size` – size (in bytes) of the MMIO memory region. It must be a multiple of the memory page size. • [in,optional] `flags` – flags that specify the MMIO memory region parameters, or `0` if these parameters are not required. The flags are defined in the header file `sysroot-*-kos/include/io/io_mmio.h` from the KasperskyOS SDK. • [out] `outRid` – pointer to the handle of the MMIO memory region. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information In the `flags` parameter, you can specify the following flags: • `MMIO_K_SHARED` – shared access with the kernel. • `MMIO_U_SHARED` – shared access with multiple registrations.

Function(s)	Information about the function(s)
`KnIoMapMem()`	Purpose Reserves the virtual memory region and maps the MMIO memory region to it. Parameters • [in] `rid` – handle of the MMIO memory region. • [in] `prot` – flags defining the access rights to the virtual memory region. The flags are defined in the header file `sysroot-*-kos/include/vmm/flags.h` from the KasperskyOS SDK. • [in] `attr` – flags defining the cache settings. The flags are defined in the header file `sysroot-*-kos/include/vmm/flags.h` from the KasperskyOS SDK. • [out] `addr` – pointer to the base address of the virtual memory region. • [out] `handle` – pointer to the handle that is used to free the virtual memory region. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information In the `prot` parameter, you can specify the following flags: • `VMM_FLAG_READ` – read access. • `VMM_FLAG_WRITE` – write access. In the `attr` parameter, you can specify the following flags: • `VMM_FLAG_WRITE_BACK`, `VMM_FLAG_WRITE_THROUGH`, `VMM_FLAG_WRITE_COMBINE`, `VMM_FLAG_CACHE_DISABLE` – cache management. It is recommended to specify the `VMM_FLAG_CACHE_DISABLE` flag.

`IoReadMmReg8()`
`IoReadMmReg16()`
`IoReadMmReg32()`
	Purpose
	
	Get an 8-, 16- or 32-bit word from an MMIO memory region.
	
	Parameters
	
	• [in] `reg` – virtual address for reading the word.
	
	Returned values
	
	8-, 16- or 32-bit word from the MMIO memory region.

`IoWriteMmReg8()`
`IoWriteMmReg16()`
`IoWriteMmReg32()`
	Purpose
	
	Write an 8-, 16- or 32-bit word to an MMIO memory region.
	
	Parameters
	
	• [in] `reg` – virtual address for writing the word.
	• [in] `data` – word to write.
	
	Returned values
	
	N/A

Function(s)	Information about the function(s)
`KnIoMemHandlesClear()`	Purpose Initializes a structure for storing the handles of resources allocated for MMIO provisioning. Parameters • [out] `handles` – pointer to the structure for storing handles. Returned values N/A Additional information After initialization, the structure fields take the `INVALID_HANDLE` value.
`KnIoMemHandlesClose()`	Purpose Closes the handles of resources allocated for MMIO provisioning and initializes the structure in which these handles were stored. Parameters • [in, out] `handles` – pointer to a structure for storing handles. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information After initialization, the structure fields take the `INVALID_HANDLE` value.

Managing interrupt processing (irq.h)

The API is defined in the header file `sysroot-*-kos/include/coresrv/io/irq.h` from the KasperskyOS SDK.

The API manages the handling of hardware interrupts. A *hardware interrupt* is a signal sent from a device to direct the processor to immediately pause execution of the current program and instead handle an event related to this device. For example, pressing a key on the keyboard invokes a hardware interrupt that ensures the required response to this pressed key (for example, input of a character).

A hardware interrupt occurs when the device queries the interrupt controller. This query can be transmitted through a hardware interrupt line between the device and the interrupt controller or through MMIO memory. In the second case, the device writes to MMIO memory by calling the *Message Signaled Interrupt* (MSI).

Each hardware interrupt line corresponds to one interrupt with a unique number.

Two types of message signaled interrupts are supported: MSI and MSI-X (extended version).

Information about API functions is provided in the table below.

Using the API

To attach an interrupt to its handler, complete the following steps:

1. Registering an interrupt.

To register an interrupt that occurs when a device queries the interrupt controller through the hardware interrupt line, call the `KnRegisterIrq()` function. If such an interrupt needs to be registered multiple times, the `IRQ_U_SHARED` flag must be specified in the `flags` parameter during each registration. If the registered interrupt is used by the kernel code, the `IRQ_K_SHARED` flag must be specified. If these flags are not specified, only the code that requests this access first (this can be either the kernel code or the code that is executed in user mode) will have access to the interrupt. All subsequent registration attempts by calling the `KnRegisterIrq()` function will end with the `rcAlreadyExists` error, and messages regarding the interrupt sharing problem will be added to the diagnostic output. If the kernel fails to access the interrupt because this interrupt is already registered by the code being executed in user mode, a message regarding the interrupt sharing problem will also be added to the diagnostic output.

To register a message signaled interrupt, call the `KnIoRegisterMsiIrq()` function after first creating a message signaled interrupt object by calling the `KnIoAllocMsi()` function. The MSI object handle can be transferred to another process via IPC. One object can be associated with only one message signaled interrupt at a time. A message signaled interrupt registration object can be reused only after closing the handle of the previously registered interrupt. After registering the message signaled interrupt, the object handle can be closed.

The data received through the `addr` and `data` output parameters of the `KnIoAllocMsi()` function must be passed to the PCIe bus controller to configure the message signaled interrupts (such as the MSI or MSI-X type) for the specified device. Moreover, one interrupt of the MSI type or one or more interrupts of the MSI-X type can be configured for one device. (The `KnIoAllocMsi()` function does not allow allocation of a sequence of interrupt vectors, thereby making it impossible to support multiple interrupts of the MSI type for one device.) If you need to configure one interrupt of the MSI or MSI-X type for a device, you must create one message signaled interrupt object via a single call of the `KnIoAllocMsi()` function. If you need to configure multiple MSI-X interrupts for a device, you must create the same number of message signaled interrupt objects via several calls of the `KnIoAllocMsi()` function. If the maximum possible number of message signaled interrupt objects have been created in the system, the `KnIoAllocMsi()` function returns `rcResourceNotFound`. If the maximum possible number of message signaled interrupt objects with the same interrupt priority have been created in the system, the `KnIoAllocMsi()` function returns `rcFail`. If the maximum number of message signaled interrupt objects have been created for the device, the `KnIoAllocMsi()` function returns `rcFail`.

An interrupt handle obtained by calling the `KnRegisterIrq()` or `KnIoRegisterMsiIrq()` function can be passed to another process via IPC.

2. Attaching a thread to an interrupt by calling the `KnIoAttachIrq()` function.

   This step is performed by the thread in whose context the interrupt will be handled.

   When using the interrupt handle obtained by calling the `KnRegisterIrq()` or `KnIoRegisterMsiIrq()` function, you can attach only one thread to an interrupt. In contrast to a message signaled interrupt, the interrupt that occurs when the device queries the interrupt controller via the hardware interrupt line can be attached to multiple threads in one or more processes. To do so, you must use different handles of the same interrupt obtained from separate calls of the `KnRegisterIrq()` function with the same flags in the `flags` parameter.

   A handle obtained by calling the `KnIoAttachIrq()` function cannot be transferred to another process via IPC.

To deny (mask) an interrupt, call the `KnIoDisableIrq()` or `KnIoDisableIrqNoSync()` function. The `KnIoDisableIrq()` function returns control after completion of the current iteration in the maskable interrupt handler except for a case in which the function is called from this handler. In this case, the `KnIoDisableIrq()` function behaves like the `KnIoDisableIrqNoSync()` function, which always returns control without waiting for completion of the current iteration in the maskable interrupt handler. In other words, the current iteration in the interrupt handler may be completed after the `KnIoDisableIrqNoSync()` function has returned control. The `KnIoDisableIrq()` function cannot be called from a critical section that is synchronized with a critical section in the maskable interrupt handler or in the handler of another interrupt.

To allow (unmask) the interrupt, call the `KnIoEnableIrq()` function. (After an interrupt is registered and a thread is attached to it, this interrupt does not require unmasking.)

Although the `KnIoDisableIrq()`, `KnIoDisableIrqNoSync()`, and `KnIoEnableIrq()` functions receive an interrupt handle through which only one thread is attached to the interrupt, their effect applies to all threads attached to this interrupt. (This does not apply to message signaled interrupts to which only one thread can be attached.)

To initiate detachment of a thread from an interrupt, call the `KnIoDetachIrq()` function outside of the thread that is attached to the interrupt. Detachment is performed by the thread attached to the interrupt by calling the `KnThreadDetachIrq()` function from the thread_api.h API.

Handling an interrupt

After attaching to an interrupt, the thread calls the `Call()` function from the syscalls.h API. The thread is locked as a result of this call. When an interrupt occurs or the `KnIoDetachIrq()` function is called, the KasperskyOS kernel sends an IPC message to the process that contains this thread. This IPC message contains a request to handle the interrupt or a request to detach the thread from the interrupt. When a process receives an IPC message, the `Call()` function in the thread attached to the interrupt returns control and provides the contents of the IPC message to the thread. The thread extracts the request from the IPC message and either processes the interrupt or detaches from the interrupt. If the interrupt is processed, information about its failure or success upon completion is added to the response IPC message that is sent to the kernel by the next `Call()` function call in the loop.

When processing an interrupt, use the `IoGetIrqRequest()` and `IoSetIrqAnswer()` functions that are declared in the header file `sysroot-*-kos/include/io/io_irq.h` from the KasperskyOS SDK. These functions let you extract data from IPC messages and add data to IPC messages for data exchange between the kernel and the thread attached to the interrupt.

The standard interrupt processing loop includes the following steps:

1. Adding information about the failure or success of interrupt processing to an IPC message by calling the `IoSetIrqAnswer()` function.

2. Sending the IPC message to the kernel and receiving an IPC message from the kernel.

   To complete this step, call the `Call()` function. In the `handle` parameter, you must specify the handle that was received when the `KnIoAttachIrq()` function was called. You must use the `msgOut` parameter to define the IPC message that will be sent to the kernel, and use the `msgIn` parameter to define the IPC message that will be received from the kernel.

3. Extracting a request from the IPC message received from the kernel by calling the `IoGetIrqRequest()` function.

4. Processing the interrupt or detaching from the interrupt depending on the request.

   If the request requires detachment from the interrupt, exit the interrupt processing loop and call the `KnThreadDetachIrq()` function from the API thread_api.h.

Deregistering an interrupt

To deregister an interrupt, complete the following steps:

1. Detach the threads from the interrupt.

   To complete this step, use the `KnThreadDetachIrq()` function from the API thread_api.h.

2. Close the handles that were received when the `KnIoAttachIrq()` function was called.

3. Close or revoke the interrupt handles in all processes that own these handles.

   In the case of a message signaled interrupt, you must additionally close or revoke the message signaled interrupt object handles in all processes that own these handles.

Information about API functions

irq.h functions

Function	Information about the function
`KnRegisterIrq()`	Purpose Registers an interrupt that occurs when a device queries an interrupt controller through a hardware interrupt line. Parameters • [in] `irq` – interrupt number. • [in] `flags` – flags defining the interrupt parameters. Flags are defined in the header files `sysroot-*-kos/include/io/io_irq.h` and `sysroot-*-kos/include/hal/irqmode.h` from the KasperskyOS SDK. • [out] `outRid` – pointer to the interrupt handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information In the `flags` parameter, you can specify the following flags: • `IRQ_LEVEL_LOW` – the interrupt occurs when the signal level is low. • `IRQ_LEVEL_HIGH` – the interrupt occurs when the signal level is high. • `IRQ_EDGE_RAISE` – the interrupt occurs when the signal level increases. • `IRQ_EDGE_FALL` – the interrupt occurs when the signal level decreases. • `IRQ_PRIO_LOW` – the interrupt has low priority. • `IRQ_PRIO_NORMAL` – the interrupt has medium priority. • `IRQ_PRIO_HIGH` – the interrupt has high priority. • `IRQ_PRIO_RT` – the interrupt has the highest priority. • `IRQ_K_SHARED` – shared access with the kernel. • `IRQ_U_SHARED` – shared access with multiple registrations.

Function	Information about the function
`KnIoAttachIrq()`	Purpose Attaches the calling thread to an interrupt. Parameters • [in] `rid` – interrupt handle. • [out] `handle` – pointer to the client IPC handle that is used by the interrupt handler. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnIoDetachIrq()`	Purpose Sends a request to a thread. When this request is fulfilled, the thread must detach from the interrupt. Parameters • [in] `rid` – interrupt handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnIoEnableIrq()`	Purpose Enables (unmasks) an interrupt. Parameters • [in] `rid` – interrupt handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnIoDisableIrq()`	Purpose Denies (masks) an interrupt, optionally waiting for completion of the current iteration in the maskable interrupt handler. Parameters • [in] `rid` – interrupt handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information The function does not wait for completion of the current iteration in the maskable interrupt handler if it is called from this handler.
`KnIoDisableIrqNoSync()`	Purpose Denies (masks) an interrupt without waiting for completion of the current iteration in the maskable interrupt handler. Parameters • [in] `rid` – interrupt handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KnIoAllocMsi()`	Purpose Creates a message signaled interrupt object. Parameters • [in] `order` – fictitious parameter (reserved for possible future use). You must specify `0`. • [in] `deviceId` – address of the device on the PCIe bus in BDF format. • [in] `priority` – interrupt priority. Constants with possible priority values are defined in the header file `sysroot-*-kos/include/io/io_irq.h` from the KasperskyOS SDK. • [out] `addr` – pointer to the MMIO memory address to which the device needs to write to trigger an interrupt (Message Address according to the PCIe specification). • [out] `data` – pointer to the data that the device needs to write to the MMIO memory at the address obtained through the `addr` parameter to trigger an interrupt (Message Data according to the PCIe specification). • [out] `msi` – pointer to the handle of the message signaled interrupt object. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information In the `priority` parameter, you can specify the following flags: • `IRQ_PRIO_LOW` – the interrupt has low priority. • `IRQ_PRIO_NORMAL` – the interrupt has medium priority. • `IRQ_PRIO_HIGH` – the interrupt has high priority. • `IRQ_PRIO_RT` – the interrupt has the highest priority.
`KnIoRegisterMsiIrq()`	Purpose Registers a message signaled interrupt. Parameters • [in] `msi` – handle of the message signaled interrupt object. • [in] `index` – fictitious parameter (reserved for possible future use). You must specify `0`. • [out] `irq` – pointer to the interrupt handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Managing threads (high-level API thread.h)

The API is defined in the header file `sysroot-*-kos/include/kos/thread.h` from the KasperskyOS SDK.

Main capabilities of the API:

- Create, terminate, and lock threads.
- Resume execution of locked threads.
- Get information about threads.

Information about API functions is provided in the table below.

> The `libkos` library also provides a low-level API for thread management. The low-level API should be used only if the high-level API does not have sufficient capabilities.

Creating a thread

To create a thread, call the `KosThreadCreate()` or `KosThreadCreateDetached()` function. These functions create a standard thread with priority ranging from 0–15. (For details about standard threads, see Managing threads (low-level API thread_api.h)).

If committing virtual memory in lazy mode is prohibited at the system level (the kernel startup parameter `vmm.overcommit=disabled` is defined) or in the process (the `KOS_TASK_FLAG_LOCK_MEM` flag is specified when creating a process using the task.h API), either virtual memory for storing the thread stack will be committed when this thread is created with the entire amount of necessary physical memory allocated, or the `KosThreadCreate()` and `KosThreadCreateDetached()` functions will end with an error if there is insufficient physical memory. (For more information about committing virtual memory, see "Allocating and freeing memory".) If committing virtual memory in lazy mode is not prohibited, the virtual memory for storing the thread stack will be committed in lazy mode.

Terminating a thread

When a thread is terminated, it is permanently deleted from the scheduler. A thread is terminated in the following cases:

- The function executed by the thread is exited.

  The root function (not a nested function) performed by the thread must be exited by the `return` operator.

- The `KosThreadTerminate()` or `KosThreadExit()` function is called.

  The `KosThreadTerminate()` function terminates the thread only if this thread was locked by calling the `KosThreadSuspend()` function or if it has been in a locked state since it was created.

  The `KosThreadExit()` function terminates the thread even if the function is called from a nested function instead of the root function executed by the thread.

- The process was terminated or switched to the "frozen" state.

  For details about the "frozen" state, see Managing processes (low-level API task_api.h).

Thread exit codes are defined by the developer. These codes should be specified in the `exitCode` parameter of the `KosThreadTerminate()` and `KosThreadExit()` functions, and when calling the `return` operator in the function executed by the thread. To get the thread exit code, call the `KosThreadWait()` function. If successful, this function returns the thread exit code. Otherwise, it returns `-1`. Therefore, thread exit codes must differ from `-1` to avoid ambiguity.

Locking and resuming a thread

To lock a thread, call the `KosThreadSuspend()` function. This function lets you lock only the calling thread.

To resume a thread that has been locked since it was created or was locked due to a call of the `KosThreadSuspend()` function, call the `KosThreadResume()` function.

To lock a thread until another thread completes, call the `KosThreadWait()` function.

To lock a thread for a defined amount of time, call the `KosThreadSleep()` function. A call of the `KosThreadSleep()` function with a 0 value for the `mdelay` parameter is identical to a call of the `KosThreadYield()` function that gives up the quantum of the calling thread to other threads until the next time the calling thread appears in the queue.

Guaranteeing that a function can only be called once

The callback function defined through the `initRoutine` parameter is called only when the `KosThreadOnce()` function is called for the first time. This does not occur on repeated calls of the `KosThreadOnce()` function (even from other threads), and the `KosThreadOnce()` function simply returns control. For example, this ensures that a driver is initialized only once when multiple software components use this driver and start its initialization irrespective of each other.

Special considerations for a thread attached to an interrupt

After a thread is attached to an interrupt, the thread becomes a real-time thread with a FIFO scheduler class and a priority higher than 31. (For details about attaching a thread to an interrupt, see Managing interrupt processing (irq.h). For details about the FIFO real-time thread scheduler class, see Managing threads (low-level API thread_api.h)).

> Some functions of this API and other APIs of the `libkos` library cannot be called from a thread that is attached to an interrupt. These functions include `KosThreadSuspend()`, `KosThreadWait()`, `KosThreadSleep()`, `KosThreadYield()`, `KnFutexWait()`, `KnNoticeGetEvent()`, `KnAuRead()`, and `KnSamplingProfiler*()`.

Getting the base address of the TLS

A *thread local storage* (TLS) is process memory in which a thread can store data in isolation from other threads. The `KosThreadTlsGet()` function gets the base address of the TLS. This function is intended for use by the `libc` library.

Getting the base address and stack limit of the thread

The `KosThreadGetStack()` function gets the base address and stack limit of the thread. This function is intended for use by the `libc` library.

Getting a TID

The KasperskyOS kernel assigns an ID to each thread. *Thread identifier* (TID) is an integer value that is unique for each thread within an entire solution. TIDs are not used to manage threads but instead are used to identify them – for example, in the task_api.h and value_api.h APIs, and also when debugging and logging programs.

To get the current thread's TID, call the `KosThreadCurrentId()` function. This function lets you get the TID only for the calling thread.

Freeing resources of a terminated thread

Resources of a thread include its stack, context, and TCB. To ensure that the resources of a thread are freed after the thread is terminated, you must close its handle before or after termination of the thread except for the following cases:

- The `KosThreadCreateDetached()` function automatically closes the handle of the created thread.
- The `KosThreadCreate()` function automatically closes the handle of the created thread if it is called with the `RTL_NULL` value in the `handle` parameter.
- The `KosThreadWait()` function closes the handle of a thread whose termination it anticipates.
- The initial thread has no handle (if this handle was not created by calling the `KnThreadOpenCurrent()` function from the thread_api.h API).

Resources of threads are also freed upon termination of the process that includes these threads.

Information about API functions

thread.h functions

Function	Information about the function
`KosThreadCreate()`	Purpose Creates a thread. Parameters • [out,optional] `handle` – pointer to the thread handle. Specifying `RTL_NULL` closes the handle automatically after the thread is created. • [in] `priority` – thread priority. • [in,optional] `stackSize` – thread stack limit (in bytes), or `0` to use the default size that was defined when the process was created. • [in] `routine` – pointer to the function executed by the thread. The parameter type is defined in the header file `sysroot-*-kos/include/coresrv/thread/thread_api.h` from the KasperskyOS SDK. • [in,optional] `context` – pointer to the parameters passed to the function defined through the `routine` parameter, or `RTL_NULL` if parameters do not need to be passed to this function. • [in] `suspended` – value defining whether the thread will be created in a suspended (`1`) or running (`0`) state. You cannot specify `1` if the `handle` parameter is set to `RTL_NULL`. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosThreadCreateDetached()`	Purpose Creates an unlocked thread and closes its handle. Parameters • [in] `priority` – thread priority. • [in,optional] `stackSize` – thread stack limit (in bytes), or `0` to use the default size that was defined when the process was created. • [in] `routine` – pointer to the function executed by the thread. The parameter type is defined in the header file `sysroot-*-kos/include/coresrv/thread/thread_api.h` from the KasperskyOS SDK. • [in,optional] `context` – pointer to the parameters passed to the function defined through the `routine` parameter, or `RTL_NULL` if parameters do not need to be passed to this function. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosThreadCurrentId()`	Purpose Gets the TID for the calling thread. Parameters N/A Returned values TID for the calling thread. The type of returned value is defined in the header file `sysroot-*-kos/include/thread/tidtype.h` from the KasperskyOS SDK.

Function	Information about the function
`KosThreadSuspend()`	Purpose Locks the calling thread. Parameters • [in] `handle` – thread handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosThreadResume()`	Purpose Resumes execution of a locked thread. Parameters • [in] `handle` – thread handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosThreadExit()`	Purpose Terminates the calling thread. Parameters • [in] `exitCode` – thread exit code. Returned values N/A
`KosThreadWait()`	Purpose Locks the calling thread until the defined thread is terminated. Parameters • [in] `handle` – thread handle. • [in] `timeout` – timeout (in milliseconds) for the termination of a thread, or `INFINITE_TIMEOUT` to define an unlimited timeout. The constant `INFINITE_TIMEOUT` is defined in the header file `sysroot-*-kos/include/rtl/rtc.h` from the KasperskyOS SDK. Returned values If successful, the function returns the thread exit code, otherwise it returns `-1`. Additional information Non-blocking call if the `timeout` parameter is set to `0`.

Function	Information about the function
`KosThreadSleep()`	Purpose Locks the calling thread for the specified duration. Parameters • [in] `mdelay` – locking time of a thread (in milliseconds), or `INFINITE_TIMEOUT` to set an unlimited locking time. The constant `INFINITE_TIMEOUT` is defined in the header file `sysroot-*-kos/include/rtl/rtc.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information Non-blocking call if the `mdelay` parameter is set to `0`.
`KosThreadYield()`	Purpose Gives up the quantum of the calling thread to other threads until the next time the calling thread appears in the queue. Parameters N/A Returned values N/A
`KosThreadTerminate()`	Purpose Terminates the locked thread. Parameters • [in] `handle` – thread handle. • [in] `exitCode` – thread exit code. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosThreadTlsGet()`	Purpose Gets the base address of the TLS for the calling thread. Parameters N/A Returned values Pointer to the TLS for the calling thread, or `RTL_NULL` if the thread does not have a TLS.
`KosThreadGetStack()`	Purpose Gets the base address and stack limit of the thread. Parameters • [in] `handle` – thread handle. • [out] `size` – pointer to the stack limit (in bytes). Returned values Pointer to the thread stack.

Function	Information about the function
`KosThreadOnce()`	Purpose Guarantees that the defined function will be called only once. Parameters • [in] `onceControl` – pointer to the variable that indicates whether or not the defined function was already called. This variable must be initialized with the `KOS_THREAD_ONCE_INIT` value. • [in] `initRoutine` – pointer to the function that must be called only one time. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Managing threads (low-level API thread_api.h)

The API is defined in the header file `sysroot-*-kos/include/coresrv/thread/thread_api.h` from the KasperskyOS SDK.

Main capabilities of the API:

- Create, terminate, and lock threads.
- Resume execution of locked threads.
- Change the priorities and scheduler classes of threads.
- Handle exceptions.
- Detach threads from interrupts.
- Attach threads to processors (processor cores).
- Get information about threads.

Information about API functions is provided in the table below.

> The `libkos` library also provides a high-level API for thread management. It is recommended to use this specific API. The low-level API should be used only if the high-level API does not have sufficient capabilities.

Creating a thread Thread scheduler classes and priorities

To create a thread, call the `KnThreadCreateByHandle()` function. You can use the `flags` parameter of this function for the created thread to define the following scheduler classes:

- Class for scheduling standard threads.
- Class for scheduling FIFO real-time threads (First In, First Out).
- Class for scheduling RR real-time threads Round-Robin).

The priority of a standard thread can take values ranging from 0 to 15. The higher the priority of a standard thread, the larger the quantum allocated to this thread and the more frequently these quanta are allocated. One standard thread with a high priority can occupy multiple consecutive places in the thread queue. Standard threads cannot supersede other standard threads or real-time threads, regardless of their respective priorities. If a real-time thread appears in the queue, the current standard thread immediately transfers control to the real-time thread. If there are no real-time threads in the queue, the current standard thread transfers control to the next standard thread if the current one is terminated, becomes locked, uses up its quantum or gives it up to the next standard thread.

The priority of a real-time thread can take values ranging from 0 to 31. Higher-priority real-time threads supersede lower-priority real-time threads. Real-time threads also supersede standard threads, regardless of their respective priorities. Control is transferred from the current real-time thread to the next real-time thread in the queue in the following cases:

- The current real-time thread with a FIFO scheduler class is terminated or locked, or a real-time thread with a higher priority appears in the queue.

  Until the real-time thread with a FIFO scheduler class is terminated, locked, or superseded by a real-time thread with a higher priority, it can retain control indefinitely and not transfer control to the next real-time thread.

- The current real-time thread with an RR scheduler class is terminated, locked or uses up its quantum, or a real-time thread with a higher priority appears in the queue.

  A real-time thread with an RR scheduler class that is not terminated, locked, or superseded by a higher-priority real-time thread transfers control to the next one upon expiry of its quantum if the next one is a real-time thread with the same priority. Otherwise, it transfers control to itself again.

- The current real-time thread has given up its quantum to the next real-time thread with the same priority in the queue.

  A real-time thread can give up its quantum to the next one in the queue if the next one is a real-time thread with the same priority. Otherwise, it transfers control to itself again.

Execution of lower-priority real-time threads begins only after each of the real-time threads with a higher priority is terminated or locked. Real-time threads with the same priority form a queue based on the FIFO principle.

In the `flags` parameter of the `KnThreadCreateByHandle()` function, you can specify the `ThreadFlagCreateStackLocked` flag, which requires that the virtual memory for storing the thread stack be committed when this thread is created with the entire required amount of physical memory allocated. (For information about committing virtual memory, see "Allocating and freeing memory".) If committing virtual memory in lazy mode is prohibited at the system level (the kernel startup parameter `vmm.overcommit=disabled` is defined) or in the process (the `KOS_TASK_FLAG_LOCK_MEM` flag is specified when creating a process using the task.h API), either the virtual memory for storing the thread stack will be committed while the entire amount of necessary physical memory will be allocated, or the `KnThreadCreateByHandle()` function will end with an error if there is insufficient physical memory, even if the `ThreadFlagCreateStackLocked` flag is not specified. If committing virtual memory in lazy mode is not prohibited, and the `ThreadFlagCreateStackLocked` flag is not specified, the virtual memory for storing the thread stack will be committed in lazy mode.

To get the thread scheduler class, call the `KnThreadGetSchedPolicyByHandle()` function.

To change the thread scheduler class, call the `KnThreadSetSchedPolicyByHandle()` function. This function also changes the size of the quantum allocated to a real-time thread with an RR scheduler class. This value is the only parameter of a real-time thread RR scheduler class that has a default value of 10 ms but can take values ranging from 2 ms to 100 ms. The scheduler class of standard threads and the scheduler class of FIFO real-time threads do not have parameters.

To get the thread priority, call the `KnThreadGetPriorityByHandle()` function.

To get the maximum and minimum possible priority of threads with the specified scheduler class, call the `KnThreadGetPriorityRange()` function.

To change the priority of a thread, call the `KnThreadSetSchedPolicyByHandle()` or `KnThreadSetPriorityByHandle()` function.

After a thread is attached to an interrupt, the thread becomes a real-time thread with a FIFO scheduler class and a priority higher than 31, irrespective of the specific scheduler class and priority this thread had before it was attached to the interrupt. Managing interrupt processing (irq.h).)

> Some API functions cannot be applied for a thread that is attached to an interrupt. These functions include `KnThreadGetPriorityByHandle()`, `KnThreadSetPriorityByHandle()`, `KnThreadGetAffinityByHandle()`, `KnThreadSetAffinityByHandle()`, `KnThreadGetSchedPolicyByHandle()`, and `KnThreadSetSchedPolicyByHandle()`. Also, some functions of this API and other APIs of the `libkos` library cannot be called from a thread that is attached to an interrupt. These functions include `KnSleep()`, `KnThreadSuspendCurrent()`, `KnThreadWaitByHandle()`, `KnFutexWait()`, `KnNoticeGetEvent()`, `KnAuRead()`, and `KnSamplingProfiler*()`.

Creating a thread handle

A thread handle is created whenever a thread is created by the `KnThreadCreateByHandle()` function call. A thread (including the initial thread) can also create its own handle by calling the `KnThreadOpenCurrent()` function.

A thread handle cannot be transferred to another process via IPC.

Handling exceptions

To register an exception handling function for a thread, call the `KnThreadSetExceptionHandler()` function. This function deregisters the previous exception handling function and returns its ID. By saving this ID, you can subsequently register the previous exception handling function again.

To get information about the last exception of a thread, call the `KnThreadGetLastException()` function in the exception handler.

Detaching a thread from an interrupt

To detach a thread from an interrupt, call the `KnThreadDetachIrq()` function. (For more details about using the `KnThreadDetachIrq()` function, see Managing interrupt processing (irq.h).)

After a thread is detached from an interrupt, the thread receives the scheduler class and priority that it had before it was attached to the interrupt.

Attaching a thread to processors (processor cores)

By default, a thread may be executed on all processors (processor cores). To restrict the set of processors (processor cores) that can be used to execute a thread, change the affinity mask of this thread. An *affinity mask* is a bit mask indicating which specific processors or processor cores must execute the thread.

To get a thread affinity mask, call the `KnThreadGetAffinityByHandle()` function.

To adjust affinity masks and perform other operations with them, use the API that is defined in the header file `sysroot-*-kos/include/rtl/cpuset.h` from the KasperskyOS SDK.

To define a thread affinity mask, call the `KnThreadSetAffinityByHandle()` function.

Terminating a thread

When a thread is terminated, it is permanently deleted from the scheduler. A thread is terminated in the following cases:

- The function executed by the thread is exited.

  The root function (not a nested function) performed by the thread must be exited by the `return` operator.

- The `KnThreadTerminateByHandle()` or `KnThreadExit()` function is called.

  The `KnThreadTerminateByHandle()` function terminates the thread only if this thread was locked by calling the `KnThreadSuspendCurrent()` function or if it has been in a locked state since it was created.

  The `KnThreadExit()` function terminates the thread even if the function is called from a nested function instead of the root function executed by the thread.

- The process was terminated or switched to the "frozen" state.

  For details about the "frozen" state, see Managing processes (low-level API task_api.h).

Thread exit codes are defined by the developer of the KasperskyOS-based solution. These codes should be specified in the `code` parameter of the `KnThreadTerminateByHandle()` and `KnThreadExit()` functions, and when calling the `return` operator in the function executed by the thread. To get the thread exit code, call the `KnThreadWaitByHandle()` function.

Locking and resuming a thread

To lock a thread, call the `KnThreadSuspendCurrent()` function.

To resume a thread that has been locked since it was created or was locked due to a call of the `KnThreadSuspendCurrent()` function, call the `KnThreadResumeByHandle()` function.

To lock a thread until another thread completes, call the `KnThreadWaitByHandle()` function.

To lock a thread for a defined amount of time, call the `KnSleep()` function.

Getting the TCB

A *thread control block* (TCB) is a structure containing thread information that is used by the kernel to manage this specific thread. The TCB contains the base address of the TLS. The `KnThreadGetTcb()` function gets the TCB address. This function is intended for use by the `libc` library.

Getting thread information

To get the current thread's TID, call the `KnThreadCurrent()` function.

The `KnThreadGetInfoByHandle()` function gets the base address and stack limit of the thread, TID, and the TLS base address. This function is intended for use by the `libc` library.

Freeing resources of a terminated thread

Resources of a thread include its stack, context, and TCB. To ensure that the resources of a thread are freed after the thread is terminated, you must close its handle before or after termination of the thread except for the following cases:

- The `KnThreadCreateByHandle()` function automatically closes the handle of the created thread if it is called with the `RTL_NULL` value in the `thread` parameter.

- The initial thread has no handle (if this handle was not created by calling the `KnThreadOpenCurrent()` function).

Resources of threads are also freed upon termination of the process that includes these threads.

Information about API functions

thread_api.h functions

Function	Information about the function
`KnThreadCreateByHandle()`	Purpose Creates a thread. Parameters • [out,optional] `thread` – pointer to the thread handle. Specifying `RTL_NULL` closes the handle automatically after the thread is created. • [in] `priority` – thread priority. • [in,optional] `stackSize` – thread stack limit (in bytes), or `0` to use the default size that was defined when the process was created. • [in] `routine` – pointer to the function executed by the thread. • [in,optional] `context` – pointer to the parameters passed to the function defined through the `routine` parameter, or `RTL_NULL` if parameters do not need to be passed to this function. • [in,optional] `flags` – flags defining the thread creation parameters or `0` to specify the default parameters, which create a standard thread in the unlocked state. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information In the `flags` parameter, you can specify the following flags: • `ThreadFlagCreateSuspended` – creates a thread in the locked state. This cannot be specified if the `thread` parameter is set to `RTL_NULL`. • `ThreadFlagCreateSchedOther` – creates a standard thread. • `ThreadFlagCreateSchedFifo` – creates a real-time thread with a FIFO scheduler class. • `ThreadFlagCreateSchedRR` – creates a real-time thread with an RR scheduler class. • `ThreadFlagCreateStackLocked` – commits the virtual memory intended for storing the thread stack while allocating the entire amount of necessary physical memory when the thread is created.
`KnThreadOpenCurrent()`	Purpose Creates the handle of the calling thread. Parameters • [out] `thread` – pointer to the thread handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KnThreadGetPriorityRange()`	Purpose Gets the maximum and minimum possible priority of threads with the specified scheduler class. Parameters • [in] `policy` – thread scheduler class. The parameter type is defined in the header file `sysroot-*-kos/include/thread/tidtype.h` from the KasperskyOS SDK. • [out,optional] `prioMin` – pointer to the minimum priority of threads, or `RTL_NULL` if this priority is not required. • [out,optional] `prioMax` – pointer to the maximum priority of threads, or `RTL_NULL` if this priority is not required. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information In the `policy` parameter, you can specify the following values: • `ThreadSchedOther` – class for scheduling standard streams. • `ThreadSchedFifo` – class for scheduling FIFO real-time threads. • `ThreadSchedRoundRobin` – class for scheduling RR real-time threads.
`KnThreadGetPriorityByHandle()`	Purpose Gets the priority of a thread. Parameters • [in] `thread` – thread handle. • [out] `priority` – pointer to the thread priority. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnThreadSetPriorityByHandle()`	Purpose Defines the priority of a thread. Parameters • [in] `thread` – thread handle. • [in] `priority` – thread priority. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnThreadSuspendCurrent()`	Purpose Locks the calling thread. Parameters N/A Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KnThreadResumeByHandle()`	Purpose Resumes execution of a locked thread. Parameters • [in] `thread` – thread handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnThreadTerminateByHandle()`	Purpose Terminates the locked thread. Parameters • [in] `thread` – thread handle. • [in] `code` – thread exit code. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnThreadExit()`	Purpose Terminates the calling thread. Parameters • [in] `code` – thread exit code. Returned values Error code.
`KnThreadGetInfoByHandle()`	Purpose Gets information about a thread. Parameters • [in] `thread` – thread handle. • [out] `info` – pointer to the structure containing the base address of the thread stack and its size (in bytes), TID, and the TLS base address. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KnThreadWaitByHandle()`	Purpose Locks the calling thread until the defined thread is terminated. Parameters • [in] `thread` – thread handle. • [in] `msDelay` – timeout (in milliseconds) for the termination of a thread, or `INFINITE_TIMEOUT` to define an unlimited timeout. The constant `INFINITE_TIMEOUT` is defined in the header file `sysroot-*-kos/include/rtl/rtc.h` from the KasperskyOS SDK. • [out] `exitCode` – pointer to the thread exit code. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Returns `rcTimeout` if the timeout has elapsed. Additional information Non-blocking call if the `msDelay` parameter is set to `0`.
`KnSleep()`	Purpose Locks the calling thread for the specified duration. Parameters • [in] `mdelay` – locking time of a thread (in milliseconds), or `INFINITE_TIMEOUT` to set an unlimited locking time. The constant `INFINITE_TIMEOUT` is defined in the header file `sysroot-*-kos/include/rtl/rtc.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information Non-blocking call if the `mdelay` parameter is set to `0`.
`KnThreadCurrent()`	Purpose Gets the TID for the calling thread. Parameters N/A Returned values TID for the calling thread. The type of returned value is defined in the header file `sysroot-*-kos/include/thread/tidtype.h` from the KasperskyOS SDK.
`KnThreadGetTcb()`	Purpose Gets the TCB address for the calling thread. Parameters N/A Returned values Pointer to the TCB for the calling thread. The data type for TCB storage is defined in the header file `sysroot-*-kos/include/thread/tcbpage.h` from the KasperskyOS SDK.

Function	Information about the function
`KnThreadDetachIrq()`	Purpose Detaches the calling thread from the interrupt handled in its context. Parameters N/A Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnThreadSetExceptionHandler()`	Purpose Registers the exception handling function for the calling thread. Parameters • [in,optional] `handler` – ID of the exception handling function, or `RTL_NULL` to deregister the previously registered exception handling function without registering a new one. The parameter type is defined in the header file `sysroot-*-kos/include/thread/tcbpage.h` from the KasperskyOS SDK. Returned values: ID of the previously registered exception handling function if one exists, otherwise `RTL_NULL`. The type of returned value is defined in the header file `sysroot-*-kos/include/thread/tcbpage.h` from the KasperskyOS SDK.
`KnThreadGetLastException()`	Purpose Gets information about the last exception of the calling thread. Parameters • [out] `exception` – pointer to the structure containing information about the exception. The type of structure is defined in the header file `sysroot-*-kos/include/thread/tcbpage.h` from the KasperskyOS SDK. Returned values N/A
`KnThreadGetAffinityByHandle()`	Purpose Gets a thread affinity mask. Parameters • [in] `thread` – thread handle. • [out] `mask` – pointer to the structure containing the thread affinity mask. The type of structure is defined in the header file `sysroot-*-kos/include/rtl/cpuset.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnThreadSetAffinityByHandle()`	Purpose Sets a thread affinity mask. Parameters • [in] `thread` – thread handle. • [in] `mask` – pointer to the structure containing the thread affinity mask. The type of structure is defined in the header file `sysroot-*-kos/include/rtl/cpuset.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KnThreadGetSchedPolicyByHandle()`	Purpose Gets information about the thread scheduler class. Parameters • [in] `thread` – thread handle. • [out] `policy` – pointer to the thread scheduler class. The data type for storing the thread scheduler class is defined in the header file `sysroot-*-kos/include/thread/tidtype.h` from the KasperskyOS SDK. • [out,optional] `param` – pointer to the parameters of the thread scheduler class, or `RTL_NULL`. The data type for storing the thread scheduler class parameters is defined in the header file `sysroot-*-kos/include/thread/tidtype.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information In the `param` parameter, you can specify `RTL_NULL` if you do not need to get the thread scheduler class parameters. In the `param` parameter, you must specify `RTL_NULL` if the thread scheduler class does not have any parameters.
`KnThreadSetSchedPolicyByHandle()`	Purpose Defines the scheduler class and priority of the thread. Parameters • [in] `thread` – thread handle. • [in] `policy` – thread scheduler class. The parameter type is defined in the header file `sysroot-*-kos/include/thread/tidtype.h` from the KasperskyOS SDK. • [in] `prio` – thread priority. • [in,optional] `param` – pointer to the parameters of the thread scheduler class, or `RTL_NULL`. The data type for storing the thread scheduler class parameters is defined in the header file `sysroot-*-kos/include/thread/tidtype.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information In the `policy` parameter, you can specify the following values: • `ThreadSchedOther` – class for scheduling standard streams. • `ThreadSchedFifo` – class for scheduling FIFO real-time threads. • `ThreadSchedRoundRobin` – class for scheduling RR real-time threads. In the `param` parameter, you can specify `RTL_NULL` if you need to define the thread scheduler class parameters that are applied by default. In the `param` parameter, you must specify `RTL_NULL` if the thread scheduler class does not have any parameters.

Managing processes (high-level API task.h)

The API is defined in the header file `sysroot-*-kos/include/kos/task.h` from the KasperskyOS SDK.

The API lets you create, start, and terminate processes, and statically create IPC channels and transfer handles.

Information about API functions is provided in the table below.

> The `libkos` library also provides a low-level API for process management. The low-level API should be used only if the high-level API does not have sufficient capabilities.

Creating a process

To create a process, call one of the following functions:

- `KosTaskCreateEx()`
- `KosTaskCreate()`
- `KosTaskInitEx()`
- `KosTaskInit()`
- `KosTaskInitFromSegEx()`
- `KosTaskInitFromSeg()`
- `KosTaskLaunch()`

With the exception of `KosTaskCreate()`, these functions receive the parameters of the created process via the input parameter `params`. This same input parameter is available in the `KosTaskSetup()` function, which lets you define the parameters of a process that was created by calling the `KosTaskCreate()` or `KosTaskCreateEx()` function. Use the `params` parameter to pass the structure containing the following fields:

- `eiid` – pointer to the process class name.
- `endpointsCount` – number of provided endpoints.

  The field can have a zero value if the process does not provide endpoints.
- `endpoints` – pointer to the array of structures containing the characteristics of provided endpoints (names and RIIDs of endpoints, names of interfaces).

  The type of structure is defined in the header file `sysroot-*-kos/include/services/handle/if_connection.h` from the KasperskyOS SDK.

  The field can have the `RTL_NULL` value if the process does not provide endpoints.

- `args` – pointer to the array of addresses of program startup parameters.

  The `RTL_NULL` value must be the last element of the array.

- `envs` – pointer to the array of addresses of program environment variables.

  Environment variables must have the format `<name>=<value>`. The `RTL_NULL` value must be the last element of the array.

- `flags` – flags defining the parameters for creating the process:

  - `KOS_TASK_FLAGS_NONE` – process creation parameters are absent.

    This flag cannot be specified together with the other flags.

  - Flags that determine the reaction to an unhandled exception in a process.

    You can specify one of the following flags:

    - `KOS_TASK_FLAG_DUMPABLE` – the process switches to the "frozen" state as a result of an unhandled exception.

      For details about the "frozen" state, see Managing processes (low-level API task_api.h).

    - `KOS_TASK_FLAG_CRITICAL` – the hardware platform is shut down or restarted as a result of an unhandled exception in the process.

      The flag may be unavailable for use depending on the specific configuration of the KasperskyOS kernel. If this flag is available, the kernel configuration also defines the specific actions to take with the hardware platform: shut down or restart.

  - Flags that determine how to commit virtual memory dynamically allocated by a process.

    For more information about committing virtual memory, see "Allocating and freeing memory".

    You can specify one of the following flags:

    - `KOS_TASK_FLAG_LOCK_MEM` – this flag prohibits committing virtual memory in lazy mode.

      Calls of functions that must set the "lazy" commit mode for a virtual memory region either commit that region and allocate the entire amount of necessary physical memory or end with an error if the required amount of physical memory cannot be allocated.

      By default, committing virtual memory in lazy mode is allowed at the system level. To prohibit committing virtual memory in lazy mode at the system level, define the kernel startup parameter `vmm.overcommit=disabled`.

    - `KOS_TASK_FLAG_OVERCOMMIT_WAIT_FREE` – if there is insufficient physical memory and a process thread queries virtual memory that must be committed in lazy mode, this thread is locked while waiting for the necessary amount of physical memory to be freed.

      This setting is applied at the system level by default. However, the flag may be required if the setting is changed at the system level but the process needs to apply the default setting.

    - `KOS_TASK_FLAG_OVERCOMMIT_EXCEPTION` – if there is insufficient physical memory and a process thread queries virtual memory that must be committed in lazy mode, an exception of the `EXCEPTION_OVERCOMMIT` type is thrown in this thread.

      The constant `EXCEPTION_OVERCOMMIT` is defined in the header file `sysroot-*-kos/include/hal/exc_codes.h` from the KasperskyOS SDK.

      For information about handling exceptions, see Managing processes (low-level API task_api.h).

      To apply this setting at the system level, you need to define the kernel startup parameter `vmm.overcommit=exception`.

- `componentTree` – pointer to the structure containing information from the formal specification of the solution component.

  The type of structure is defined in the header file `sysroot-*-kos/include/nk/types.h` from the KasperskyOS SDK. This structure is an element of automatically generated transport code (structure named `*_component_tree`).

  The field may have the value `RTL_NULL`.

- `stackSize` – default thread stack limit, in bytes. If `0` is set, the stack will be 1 MB by default.

When setting process parameters via the input parameter `params` of the `KosTaskCreateEx()`, `KosTaskSetup()`, `KosTaskInit*()`, and `KosTaskLaunch()` functions, you must take into account the following details:

- The `KosTaskCreateEx()` function uses only the `flags` and `stackSize` fields.
- The `KosTaskSetup()` function ignores the `eiid` and `flags` fields, and the process parameters corresponding to these fields are defined when calling the `KosTaskCreate*()` functions.
- The `KosTaskSetup()` function ignores the `stackSize` field.
- The `KosTaskCreate()`, `KosTaskInit()`, `KosTaskInitEx()`, and `KosTaskLaunch()` functions write `0` in the `stackSize` field.
- If the `componentTree` field has a value other than `RTL_NULL`, the `eiid` field will be ignored by the `KosTaskInit*()` and `KosTaskLaunch()` functions, and the process class name will be taken from the formal specification of the solution component.
- If the `componentTree` field has a value other than `RTL_NULL`, the `endpointsCount` and `endpoints` fields will be ignored by the `KosTaskSetup()`, `KosTaskInit*()`, and `KosTaskLaunch()` functions, and parameters of the provided endpoints (including the parameters of endpoints of embedded components) will be taken from the formal specification of the solution component.

When the `KosTaskInit()` or `KosTaskLaunch()` function is called, one of the following values is used as the process name and executable file name:

- Value of the `eiid` field, if the value of the `componentTree` field is equal to `RTL_NULL`.
- Name of the process class from the formal specification of the solution component, if the `componentTree` field value is different from `RTL_NULL`.

Likewise, these values are applied as the process name and/or executable file path if you call the `KosTaskInitEx()` or `KosTaskInitFromSegEx()` function with the `RTL_NULL` value in the `name` parameter and/or the `path` parameter. These values are also applied as the process name if you call the `KosTaskInitFromSeg()` function with the `RTL_NULL` value in the `name` parameter.

The `KosTaskCreate*()` and `KosTaskInit*()` functions use the `outTask` output parameter to pass the address of the object describing the child process. This object is a structure that is created in the memory of the parent process and the child process. The developer of a solution does not need to perform operations with the fields of this structure. However, the pointer to this structure must be used as the process ID when calling API functions. A child process can get the address of the object describing it by calling the `KosTaskGetSelf()` function.

If statically created IPC channels are used to access endpoints provided by a server process, the object describing this server process must be linked to the structures containing information about endpoints from the formal specification of the solution component. This is necessary so that client processes can receive information about endpoints provided by the server process when creating a static IPC channel. To link an object describing a child server process to structures containing information about endpoints from the formal specification of the solution component, this information must be passed through the `componentTree` field of the `params` parameter when calling the `KosTaskSetup()`, `KosTaskInit*()`, and `KosTaskLaunch()` functions. A server process that is already running can link the object describing it to structures containing information about endpoints from the formal specification of the solution component by calling the `KosTaskSetComponentTree()` function. This is required if the running server process does not have a parent process.

When the `KosTaskInitEx()`, `KosTaskInit()` or `KosTaskLaunch()` function is called, the ELF image from the defined executable file in ROMFS is loaded into the memory of the created process. If the ELF image contains the symbol table `.symtab` and string table `.strtab`, they are loaded into process memory. Using these tables, the kernel receives the names of functions for generating stack backtrace data (call stack information).

To get information about the ELF image loaded into process memory, call the `KosTaskGetElfSegs()` function. This function is applicable only to those processes that were created with the specified `KOS_TASK_FLAG_DUMPABLE` flag. When using this function, the symbol table `.symtab` and string table `.strtab` are considered to be loadable ELF image segments along with segments of the `LOAD` type.

In contrast to the `KosTaskInit*()` and `KosTaskLaunch()` functions, the `KosTaskCreate*()` functions create an *"empty" process*, which is a process in whose memory no program ELF image has been loaded. To load ELF image segments into process memory, call the `KosTaskLoadSegments()` function. To use the `KosTaskLoadSegments()` function and the `KosTaskInitFromSegEx()` function, MDL buffers containing loadable ELF image segments must be created in advance. Loadable ELF image segments must be defined via the `segs` parameter. In addition, the symbol table `.symtab` and the string table `.strtab` are also considered to be loadable segments along with segments of the `LOAD` type. You must use the `entry` and `relocBase` parameters to define the program entry point and the ELF image load offset, respectively. The program entry point is the sum of the address specified in the `e_entry` field of the ELF image header and the ELF image load offset. (For an ELF file of the `DYN` type, the ELF image load offset must be a random value for ASLR support.) To get the ELF image load offset, you can call the `KnTaskVmReserveForElf()` function from the task_api.h API, which reserves the virtual memory region for loading the ELF image. If the ELF file has the `DYN` type, the `KnTaskVmReserveForElf()` function uses the `relocBase` output parameter to pass a random offset, and it passes a zero offset for an ELF file of a different type. (The `KosTaskInitEx()`, `KosTaskInit()`, and `KosTaskLaunch() functions` also set a random offset for an ELF file of the `DYN` type and a zero offset for an ELF file of a different type.) Data passed to the `KosTaskLoadSegments()` and `KosTaskInitFromSegEx()` functions with the `count`, `segs`, and `entry` parameters, and parameters related to loading the symbol table `.symtab` and the string table `.strtab` are prepared by the `KnElfCreateVmSegEx()` function from the elf_api.h API.

The `KosTaskInitFromSeg()` function is a simplified version of the `KosTaskInitFromSegEx()` function and does not let you load the symbol table `.symtab`, string table `.strtab` into process memory, and does not let you define the ELF image load offset (instead, it sets a null offset).

The `KosTaskLaunch()` function creates and immediately starts a process without the capability to statically create IPC channels.

The `KosTaskSetArgs()` and `KosTaskSetEnv()` functions define the program startup parameters and environment variables, respectively. These functions are intended for use by the `libkos` library.

The `KosTaskAddEndpoints()` function registers the endpoints provided by the process. This function is intended for use by the `libkos` library.

Statically created IPC channels

Before starting processes, you can create IPC channels between them. You can create multiple IPC channels with different names between one client process and one server process. You can create IPC channels with the same name between one server process and multiple client processes.

To create an IPC channel with a name matching the name of a server process class, call the `KosTaskConnect()` function.

To create an IPC channel with a defined name, call the `KosTaskConnectToService()` function.

To use the created IPC channel, you must get the IPC handle on the client side and on the server process side by calling the `SLStaticResolveHandle()` function from the API sl-static.h. This function uses the `channelName` parameter to receive the name of the IPC channel.

Transferring handles to a child process

A parent process can transfer handles to a child process that is not yet running. (General information about transferring handles is provided in the Transferring handles section.)

To pass a handle to a child process, call the `KosTaskTransferResource()` function while specifying the handle of the resource transfer context object, the permissions mask, and the conditional name of the descendant of the transferred handle in addition to the other parameters.

To find the descendant of a handle transferred by a parent process, call the `KosTaskLookupResource()` function from the child process while specifying the conditional name of the descendant of the handle that was defined by the parent process when calling the `KosTaskTransferResource()` function.

Starting a process

To create and immediately start a process without statically creating IPC channels, call the `KosTaskLaunch()` function.

To start an already created process for which you can create the necessary IPC channels before starting this process, call the `KosTaskRunEx()` or `KosTaskRun()` function.

Use the `fsBackend` parameter of the `KosTaskRunEx()` function to specify whether to use the kernel or the system program `FSUsr` to support the ROMFS file system for the started process by default (for details, see Using the ROMFS file system (low-level API fs_api.h)). In a process started by a call of the `KosTaskRun()` or `KosTaskLaunch()` function, the default support of the ROMFS file system is the same as the default support for the parent process when the `KosTaskRun()` or `KosTaskLaunch()` is called. To get information about how the ROMFS file system is supported for a process, call the `KosTaskGetSelfFSBackend()` function from this process. (The `KosTaskSetSelfFSBackend()` function writes information regarding how the ROMFS file system is supported for the calling process to the object describing the process. This function is intended for use by the `libkos` library.)

The parent process can put the ROMFS file system in the user space and mount it so that it can then be used by the child process by default. To do so, the parent process calls the `KnFsChangeBackend()` function from the fs_api.h API to indicate that the ROMFS file system for the process is supported by the `FSUsr` program, and it calls the `KnFsChange()` function from the fs_api.h API to mount the required ROMFS image. Then the parent process starts the child process by calling the `KosTaskRun()`, `KosTaskRunEx()` or `KosTaskLaunch()` function. (When calling the `KosTaskRunEx()` function, you must explicitly specify that the ROMFS file system for the child process will be supported by the `FSUsr` program.) As a result, by default the child process will use the ROMFS file system that is put into the user space by the parent process.

If the parent process does not need to terminate the child process or wait for its termination, the object describing the child process must be deleted and the reference counter for it must be reset by using the `KosTaskPut()` function after the child process is started. The `KosTaskLaunch()` function calls the `KosTaskPut()` function after the child process is started.

Terminating a process

The API terminates and waits for termination of child processes.

To terminate a child process, call the `KosTaskStop()` or `KosTaskStopAndWait()` function.

To wait for a child process to terminate on its own initiative, call the `KosTaskWait()` function.

To ensure that the kernel object describing a child process is deleted after this process is terminated, its handle in the parent process must be closed before or after termination of the child process. The handle of the child process is closed when the object describing the child process is deleted from the memory of the parent process. To delete an object describing a child process, reset the reference counter for this object by using the `KosTaskPut()` function. The `KosTaskLaunch()` function calls the `KosTaskPut()` function after the child process is started.

For details about terminating processes, see Managing processes (low-level API task_api.h).

Getting a process handle

To get the handle of a process, call the `KosTaskGetHandle()` function.

The handle of a process can be transferred to another process via IPC.

The handle of a calling process that was received by calling the `KosTaskGetHandle()` function cannot be closed. If closed, this handle may be used to identify other resources, which will lead to errors. Descendants of this handle can be closed without a risk of similar errors occurring.

Getting the startup parameters and environment variables of a program

The `KosTaskGetArgsList()` function lets you get the startup parameters of a program. The `KosTaskFreeArgsList()` function lets you delete the array of addresses of program startup parameters that were obtained by calling the `KosTaskGetArgsList()` function.

The `KosTaskGetEnvList()` function lets you get the environment variables of a program. The `KosTaskFreeEnvList()` function lets you delete the array of addresses of program environment variables that were obtained by calling the `KosTaskGetEnvList()` function.

All these functions are intended for use by the `libc` library.

Information about API functions

task.h functions

Function	Information about the function
`KosTaskCreate()`	Purpose Creates a process. Parameters • [in] `name` – pointer to the process name. • [in] `eiid` – pointer to the process class name. • [in] `path` – pointer to the executable file path. This function does not use the specified path to query the executable file but instead only saves it as metadata of the created process. • [in] `flags` – flags defining the parameters for creating the process. • [out] `outTask` – pointer to the address of the object describing the process. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosTaskCreateEx()`	Purpose Creates a process. Parameters • [in] `name` – pointer to the process name. • [in] `eiid` – pointer to the process class name. • [in] `path` – pointer to the executable file path. This function does not use the specified path to query the executable file but instead only saves it as metadata of the created process. • [in] `params` – pointer to the structure containing process parameters. • [out] `outTask` – pointer to the address of the object describing the process. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosTaskSetup()`	Purpose Sets the parameters of a process. Parameters • [in] `task` – pointer to the object that describes the process. • [in] `params` – pointer to the structure containing process parameters. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KosTaskLoadSegments()`	Purpose Loads ELF image segments into process memory. Parameters • [in] `task` – pointer to the object that describes the process. • [in] `count` – number of loaded ELF image segments. • [in] `relocBase` – ELF image load offset (in bytes). • [in] `entry` – address of the program entry point. • [in,out] `segs` – pointer to the array of structures describing loaded ELF image segments. The type of structure is defined in the header file `sysroot-*-kos/include/coresrv/vmm/vmm_types.h` from the KasperskyOS SDK. If the `loadAddr` field of this structure is set equal to `0`, the load address of the segment will be selected automatically and written to the `loadAddr` field. • [in,optional] `symTabIndex` – index of loaded the ELF image segment containing the symbol table `.symtab`. Enumeration starts with zero. The parameter value is ignored if at least one of the `symTabSize` and `strTabSize` parameters has a zero value. • [in,optional] `symTabSize` – size of the symbol table `.symtab`, or `0` if you do not need to load this table. If this parameter has a zero value, the `strTabSize` parameter must also have a zero value. • [in,optional] `strTabIndex` – index of the loaded ELF image segment containing the string table `.strtab`. Enumeration starts with zero. The parameter value is ignored if at least one of the `symTabSize` and `strTabSize` parameters has a zero value. • [in,optional] `strTabSize` – size of the string table `.strtab` (in bytes), or `0` if you do not need to load this table. If this parameter has a zero value, the `symTabSize` parameter must also have a zero value. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosTaskInitEx()`	Purpose Creates a process. Parameters • [in] `params` – pointer to the structure containing process parameters. • [in,optional] `name` – pointer to the process name, or `RTL_NULL` to use the process class name, received via the `params` parameter, as the process name. • [in,optional] `path` – pointer to the name of the executable file in ROMFS, or `RTL_NULL` to use the process class name, received via the `params` parameter, as the executable file name. • [out] `outTask` – pointer to the address of the object describing the process. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosTaskInit()`	Purpose Creates a process. Parameters • [in] `params` – pointer to the structure containing process parameters. • [out] `outTask` – pointer to the address of the object describing the process. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KosTaskInitFromSegEx()`	Purpose Creates a process. Parameters • [in] `params` – pointer to the structure containing process parameters. • [in,optional] `name` – pointer to the process name, or `RTL_NULL` to use the process class name, received via the `params` parameter, as the process name. • [in,optional] `path` – pointer to the executable file path or `RTL_NULL` to use the process class name, received via the `params` parameter, as the executable file path. This function does not use the specified path to query the executable file but instead only saves it as metadata of the created process. • [in] `count` – number of loaded ELF image segments. • [in] `relocBase` – ELF image load offset (in bytes). • [in] `entry` – address of the program entry point. • [in,out] `segs` – pointer to the array of structures describing loaded ELF image segments. The type of structure is defined in the header file `sysroot-*-kos/include/coresrv/vmm/vmm_types.h` from the KasperskyOS SDK. If the `loadAddr` field of this structure is set equal to `0`, the load address of the segment will be selected automatically and written to the `loadAddr` field. • [in,optional] `symTabIndex` – index of loaded the ELF image segment containing the symbol table `.symtab`. Enumeration starts with zero. The parameter value is ignored if at least one of the `symTabSize` and `strTabSize` parameters has a zero value. • [in,optional] `symTabSize` – size of the symbol table `.symtab`, or `0` if you do not need to load this table. If this parameter has a zero value, the `strTabSize` parameter must also have a zero value. • [in,optional] `strTabIndex` – index of the loaded ELF image segment containing the string table `.strtab`. Enumeration starts with zero. The parameter value is ignored if at least one of the `symTabSize` and `strTabSize` parameters has a zero value. • [in,optional] `strTabSize` – size of the string table `.strtab` (in bytes), or `0` if you do not need to load this table. If this parameter has a zero value, the `symTabSize` parameter must also have a zero value. • [out] `outTask` – pointer to the address of the object describing the process. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosTaskInitFromSeg()`	Purpose Creates a process. Parameters • [in] `params` – pointer to the structure containing process parameters. • [in,optional] `name` – pointer to the process name, or `RTL_NULL` to use the process class name, received via the `params` parameter, as the process name. • [in] `count` – number of loaded ELF image segments. • [in] `entry` – address of the program entry point. • [in,out] `segs` – pointer to the array of structures describing loaded ELF image segments. The type of structure is defined in the header file `sysroot-*-kos/include/coresrv/vmm/vmm_types.h` from the KasperskyOS SDK. If the `loadAddr` field of this structure is set equal to `0`, the load address of the segment will be selected automatically and written to the `loadAddr` field. • [out] `outTask` – pointer to the address of the object describing the process. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KosTaskGetElfSegs()`	Purpose Gets information about the ELF image loaded in process memory. Parameters • [in] `task` – pointer to the object that describes the process. • [out] `segsBuf` – pointer to the array used to save structures describing loaded ELF image segments. The type of structure is defined in the header file `sysroot-*-kos/include/coresrv/vmm/vmm_types.h` from the KasperskyOS SDK. • [in] `segsBufSize` – size (in bytes) of the buffer used to store the array of structures that describe the loaded ELF image segments. • [out] `segsCount` – pointer to the number of loaded ELF image segments. • [out] `symTabIndex` – pointer to the index of the loaded ELF image segment containing the symbol table `.symtab`. • [out] `symTabSize` – pointer to the size of the symbol table `.symtab` (in bytes). • [out] `strTabIndex` – pointer to the index of the loaded ELF image segment containing the string table `.strtab`. • [out] `strTabSize` – pointer to the size of the string table `.strtab` (in bytes). Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosTaskConnect()`	Purpose Creates an IPC channel. Parameters • [in] `cl` – pointer to the object that describes the client process. • [in] `sr` – pointer to the object that describes the server process. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosTaskConnectToService()`	Purpose Creates an IPC channel with the defined name. Parameters • [in] `cl` – pointer to the object that describes the client process. • [in] `sr` – pointer to the object that describes the server process. • [in] `name` – pointer to the name of the IPC channel. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KosTaskSetArgs()`	Purpose Defines the program startup parameters. Parameters • [in] `task` – pointer to the object that describes the process. • [in] `args` – pointer to an array of addresses of program startup parameters. The `RTL_NULL` value must be the last element of the array. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosTaskGetArgsList()`	Purpose Gets the program startup parameters. Parameters • [in] `task` – pointer to the object that describes the process. • [out] `args` – pointer to the address of the array of program startup parameter addresses, or `RTL_NULL` if these parameters are not available. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosTaskFreeArgsList()`	Purpose Deletes the array of addresses of program startup parameters that was received by calling the `KosTaskGetArgsList()`. Parameters • [in] `args` – pointer to an array of addresses of program startup parameters. Returned values N/A
`KosTaskSetEnv()`	Purpose Defines environment variables of a program. Parameters • [in] `task` – pointer to the object that describes the process. • [in] `envs` – pointer to the array of addresses of program environment variables. The `RTL_NULL` value must be the last element of the array. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosTaskGetEnvList()`	Purpose Gets the environment variables of a program. Parameters • [in] `task` – pointer to the object that describes the process. • [out] `envs` – pointer to the address of the array of program environment variable addresses, or `RTL_NULL` if these variables are not available. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KosTaskFreeEnvList()`	Purpose Deletes the array of program environment variable addresses that was received by calling the `KosTaskGetEnvList()` function. Parameters • [in] `envs` – pointer to the array of addresses of program environment variables. Returned values N/A
`KosTaskRunEx()`	Purpose Resumes a process. Parameters • [in] `task` – pointer to the object that describes the process. • [in] `fsBackend` – parameter that specifies whether the ROMFS file system for the started process is supported by the kernel or by the system program `FSUsr` by default. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosTaskRun()`	Purpose Resumes a process. Parameters • [in] `task` – pointer to the object that describes the process. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosTaskStop()`	Purpose Terminates a process. Parameters • [in] `task` – pointer to the object that describes the process. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosTaskStopAndWait()`	Purpose Terminates a process and waits for the termination of this process. Parameters • [in] `task` – pointer to the object that describes the process. • [in] `timeout` – timeout (in milliseconds), or `INFINITE_TIMEOUT` to define an unlimited timeout. The constant `INFINITE_TIMEOUT` is defined in the header file `sysroot-*-kos/include/rtl/rtc.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information Non-blocking call if the `timeout` parameter is set to `0`.

Function	Information about the function
`KosTaskWait()`	Purpose Waits for the termination of a process. Parameters • [in] `task` – pointer to the object that describes the process. • [in] `timeout` – timeout (in milliseconds), or `INFINITE_TIMEOUT` to define an unlimited timeout. The constant `INFINITE_TIMEOUT` is defined in the header file `sysroot-*-kos/include/rtl/rtc.h` from the KasperskyOS SDK. • [out] `retcode` – pointer to the process exit code. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information Non-blocking call if the `timeout` parameter is set to `0`.
`KosTaskGetHandle()`	Purpose Gets the process handle. Parameters • [in] `task` – pointer to the object that describes the process. Returned values Process handle.
`KosTaskLaunch()`	Purpose Creates and starts a process. Parameters • [in] `params` – pointer to the structure containing process parameters. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosTaskGetSelf()`	Purpose Gets the address of an object that describes a calling process. Parameters N/A Returned values Pointer to the object describing the process, or `RTL_NULL` if this object was deleted. Additional information The function adds 1 to the reference counter for the object describing the process, therefore the `KosTaskPut()` function must be called after this function is called.

Function	Information about the function
`KosTaskPut()`	Purpose Subtracts 1 from the reference counter for the object describing the process. Parameters • [in] `task` – pointer to the object that describes the process. Returned values N/A
`KosTaskAddEndpoints()`	Purpose Registers endpoints. Parameters • [in,out] `task` – pointer to the object that describes the process. • [in] `endpoints` – pointer to the array of structures describing endpoints. The type of structure is defined in the header file `sysroot-*-kos/include/services/handle/if_connection.h` from the KasperskyOS SDK. • [in] `count` – number of endpoints. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosTaskSetSelfFSBackend()`	Purpose Writes information about how ROMFS file system support is provided for the calling process (by the kernel or by the `FSUsr` system program) to the object describing the process. Parameters • [in] `fsBackend` – parameter that specifies whether the ROMFS file system is supported by the kernel or by the system program `FSUsr`. Returned values N/A
`KosTaskGetSelfFSBackend()`	Purpose Gets information indicating whether the ROMFS file system for a calling process is supported by the kernel or by the system program `FSUsr`. Parameters N/A Returned values Value indicating whether the ROMFS file system is supported by the kernel or by the system program `FSUsr`.

Function	Information about the function
`KosTaskSetComponentTree()`	Purpose Creates structures containing information about endpoints from the formal specification of the solution component and links these structures to the object describing the process. Parameters • [in] `task` – pointer to the object that describes the process. • [in] `root` – pointer to the structure containing information from the formal specification of the solution component. The type of structure is defined in the header file `sysroot-*-kos/include/nk/types.h` from the KasperskyOS SDK. This structure is an element of automatically generated transport code. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosTaskTransferResource()`	Purpose Transfers a handle to a process that is not yet running. Parameters • [in] `task` – pointer to the object that describes the process. • [in] `srcHandle` – transferred handle. • [in] `srcBadge` – handle of the resource transfer context object. • [in] `dstRights` – permissions mask of the descendant of the transferred handle. • [in] `handleName` – pointer to the name of the descendant of the transferred handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosTaskLookupResource()`	Purpose Searches for a handle based on the name defined by the `KosTaskTransferResource()` function. Parameters • [in] `handleName` – pointer to the handle name. Returned values If successful, it returns the handle. Otherwise, it returns `INVALID_HANDLE`.

Managing processes (low-level API task_api.h)

The API is defined in the header file `sysroot-*-kos/include/coresrv/task/task_api.h` from the KasperskyOS SDK.

Main capabilities of the API:

- Create, start, and terminate processes.
- Handle exceptions.
- Get information about processes, including information about the reasons for their termination.
- Set priorities and scheduling classes for initial process threads.

Information about API functions is provided in the table below.

> The `libkos` library also provides a high-level API for process management. It is recommended to use this specific API. The low-level API should be used only if the high-level API does not have sufficient capabilities.

Creating and starting a process

The following API functions are required for creating and running a process:

- `KnTaskCreate()` or `KnTaskCreateEx()`
- `KnTaskReseedAslr()`
- [optional] `KnTaskVmReserveForElf()`
- `KnTaskLoadSeg()`
- [optional] `KnTaskLoadElfSyms()`
- `KnTaskSetInitialState()`
- `KnTaskSetEnv()`
- `KnTaskResume()`

These functions are intended for use by the `libkos` library. The solution developer must use the task.h API functions to create and run processes. For many functions of the task_api.h API that are intended for use by the solution developer, the input parameter is the process handle. To get the handle of a process, call the `KosTaskGetHandle()` function from the task.h API. The handle of a process can be transferred to another process via IPC.

The `KnTaskVmReserveForElf()` function is intended for use both by the `libkos` library and by the solution developer. This function reserves a region of virtual memory for loading the ELF image. The `task` and `elfDescr` input parameters are used to accept the process handle and the pointer to the object describing the ELF image, respectively. (The object describing the ELF image is created by the `KnElfOpen()` function from the elf_api.h API) The `relocBase` output parameter is used to pass the ELF image load offset. If the ELF file has the `DYN` type, it passes a random offset. It passes a zero offset for an ELF file of a different type. The ELF image load offset is an input parameter of the `KosTaskLoadSegments()` and `KosTaskInitFromSegEx()` functions from the task.h API and of the `KnElfCreateVmSegEx()` function from the elf_api.h API. A random ELF image load offset is part of the address space layout randomization (ASLR) support. *Address Space Layout Randomization* Address Space Layout Randomization (ASLR) is the use of random addresses for the location of process memory regions occupied by an ELF image, dynamic libraries, and the stack and heap of a program to make it more difficult to exploit vulnerabilities associated with a process address space structure that is known by a hacker in advance.

Before a process is started, it receives data from its parent process via a static connection page. A *static connection page* (SCP) is a set of structures containing data for statically creating IPC channels, startup parameters, and environment variables of a program. A parent process writes data to the SCP of a child process by calling the `KnTaskSetEnv()` function. When a child process is started, it reads data from the SCP by calling the `KnTaskGetEnv()` function, then it deletes the SCP by calling the `KnTaskFreeEnv()` function. Like the `KnTaskSetEnv()` function, the `KnTaskGetEnv()` and `KnTaskFreeEnv()` functions are intended for use by the `libkos` library.

By default, the initial thread of a process is a standard thread. (For details about thread scheduler classes, see Managing threads (low-level API thread_api.h) To get the priority of the initial thread of a process, call the `KnTaskGetInitialThreadPriority()` function. To change the priority of the initial thread of a process, call the `KnTaskSetInitialThreadPriority()` function. To change the scheduler class and/or priority of the initial thread of a process, call the `KnTaskSetInitialPolicy()` function. All three functions can be used after setting the program entry point and after starting the process. The program entry point is set by all the process creation functions from the task.h API, except for `KosTaskCreate()`. If the process is created by the `KosTaskCreate()` function, the entry point will be set after calling the `KosTaskLoadSegments()` function.

Terminating a process

Process termination includes the following:

- Terminating all threads of the process.
- Freeing the memory of the process.
- Freeing system and user resources that are exclusively owned by the process.

  When a process is terminated, all the handles that it owns are closed. If a closed handle was the only handle of a resource, this resource is freed.

A process can be terminated for the following reasons:

- On its own initiative.

  The `KnTaskExit()` function is called, or all threads of the process are terminated.
- By external request.

  The `KnTaskTerminate()` function is called.
- As a result of an unhandled exception (crash).

  By calling the `KnTaskPanic()` function, a process can purposefully initiate an exception that cannot be handled and leads to the process terminating. This assures that the process can terminate itself. (For example, if the process contains threads attached to interrupts, the `KnTaskExit()` function cannot terminate this process but the `KnTaskPanic()` function can.)

Process exit codes are defined by the solution developer. These codes must be specified in the `status` parameter of the `KnTaskExit()` function. If a process was terminated due to the termination of all its threads, the exit code of this process will be the exit code of its initial thread. To get the exit code of a process that was terminated on its own initiative, call the `KnTaskGetExitCode()` function.

To get information regarding the reason for process termination, call the `KnTaskGetExitStatus()` function. This function determines whether a process was terminated on its own initiative, by external request, or unexpectedly.

To get information about an unhandled exception that led to an unexpected termination of a process, call the `KnTaskGetExceptionInfo()` function. This information includes the exception code and the context of the thread in which this exception occurred.

If the process was created by one of the functions of the task.h API with the specified `KOS_TASK_FLAG_DUMPABLE` flag, an unhandled exception will cause this process to go into a "frozen" state and the process will not terminate. When a process is switched to a *"frozen" state*, its threads are terminated as a result of the unhandled exception but its resources are not freed, which means that you can get information about this process. To get the context of a thread that is part of a frozen process, call the `KnTaskGetThreadContext()` function. To get information about the virtual memory region that belongs to a frozen process, call the `KnTaskGetNextVmRegion()` function. This information includes the base address and size of the virtual memory region, and the access rights to this virtual memory region. Before a process switches to the frozen state, stack backtrace data (call stack information) is printed for the thread in which the unhandled exception occurred. To terminate a frozen process, call the `KnTaskTerminateAfterFreezing()` function.

To ensure that the kernel object describing a process is deleted after this process is terminated, each of its handles must be closed or revoked in all processes that own these handles before or after this process is terminated.

Handling exceptions

To register an exception handling function for a process, call the `KnTaskSetExceptionHandler()` function from this process. This function deregisters the previous exception handling function and returns its ID. By saving this ID, you can subsequently register the previous exception handling function again.

An exception handling function is called when an exception occurs in any process thread. If the exception is successfully handled, this function returns a value other than zero. Otherwise it returns zero. The input parameter of the exception handling function is a structure containing information about the exception. The type of this structure is defined in the header file `sysroot-*-kos/include/thread/tcbpage.h` from the KasperskyOS SDK.

If an exception handling function registered at the process level failed to successfully handle an exception, the exception handling function registered at the level of the thread in which the exception occurred will be called. (For details about handling exceptions at the thread level, see Managing threads (low-level API thread_api.h)).

Receiving information about a process

To get the process ID (PID), call the `KnTaskGetId()` or `KnTaskGetIdByHandle()` function. The `KnTaskGetId()` function lets you get the PID of the calling process. The `KnTaskGetIdByHandle()` function lets you get the PID of a process based on its handle.

To get the name of a process, call the `KnTaskGetName()` function from this process.

To get the path to the executable file from which the process was created, call the `KnTaskGetPath()` function from this process.

Transferring handles to a child process

A parent process can transfer handles to a child process that is not yet running. (General information about transferring handles is provided in the Transferring handles section.) The `KnTaskTransferResource()` function lets you transfer a handle to a child process. This function is intended for use by the `libkos` library. The solution developer needs to use the `KosTaskTransferResource()` function from the task.h API, which not only transfers the handle, but also copies it to the SCP of the child process so that the child process can find the transferred handle by calling the `KosTaskLookupResource()` function from the task.h API.

Reserving memory in a child process

The API includes the `KnTaskVmReserve()` and `KnTaskVmFree()` functions, which reserve and free virtual memory regions, respectively, in a child process that does not yet have a defined program entry point. These functions are intended for use by the `libkos` library.

Getting the GSI address

Global system information (GSI) is a structure containing system information, such as the timer count since the kernel started, the timer counts per second, the number of processors (processor cores) in active state, and processor cache data. The type of structure is defined in the header file `sysroot-*-kos/include/task/pcbpage.h` from the KasperskyOS SDK. The `KnTaskGetGsi()` function gets the GSI address. This function is intended for use by the `libc` library.

Getting the PCB address

A *process control block* (PCB) is a structure containing process information that is used by the kernel to manage this process. The type of structure is defined in the header file `sysroot-*-kos/include/task/pcbpage.h` from the KasperskyOS SDK. The `KnTaskGetPcb()` function gets the PCB address. This function is intended for use by the `libc` library.

Getting the addresses and sizes of the symbol table.symtab and the string table .strtab

The `KnTaskGetElfSyms()` function lets you get the addresses and sizes of the symbol table `.symtab` and the string table `.strtab` loaded into the process memory. This function is intended for a mechanism that prints stack backtrace data when operating in user mode (not in kernel mode).

Information about API functions

task_api.h functions

Function	Information about the function
`KnTaskCreate()`	Purpose Creates a process. Parameters • [in] `name` – pointer to the process name. • [in] `eiid` – pointer to the process class name. • [in,optional] `path` – pointer to the executable file path, or `RTL_NULL` if you do not need to specify the path to the executable file. This function does not use the specified path to query the executable file but instead only saves it as metadata of the created process. • [in,optional] `stackSize` – thread stack limit (in bytes) used by default when creating process threads. If `0` is specified, the stack will be 1 MB by default. • [in] `priority` – priority of the initial thread. The parameter type is defined in the header file `sysroot-*-kos/include/coresrv/thread/thread_api.h` from the KasperskyOS SDK. • [out] `outHandle` – pointer to the process handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KnTaskCreateEx()`	Purpose Creates a process. Parameters • [in] `name` – pointer to the process name. • [in] `eiid` – pointer to the process class name. • [in,optional] `path` – pointer to the executable file path, or `RTL_NULL` if you do not need to specify the path to the executable file. This function does not use the specified path to query the executable file but instead only saves it as metadata of the created process. • [in,optional] `stackSize` – thread stack limit (in bytes) used by default when creating process threads. If `0` is specified, the stack will be 1 MB by default. • [in] `priority` – priority of the initial thread. The parameter type is defined in the header file `sysroot-*-kos/include/coresrv/thread/thread_api.h` from the KasperskyOS SDK. • [in] `flags` – flags defining the parameters for creating the process. • [out] `outHandle` – pointer to the process handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information In the `flags` parameter, you can specify the following flags: • Flags that determine the reaction to an unhandled exception. One of the following flags must be specified: ◦ `TaskExceptionTerminatesTask` – the process is terminated as a result of an unhandled exception. ◦ `TaskExceptionFreezesTask` – the process switches to the "frozen" state as a result of an unhandled exception. ◦ `TaskExceptionTerminatesSystem` – the hardware platform is shut down or restarted as a result of an unhandled exception in the process. The `TaskExceptionTerminatesSystem` flag may be unavailable for use depending on the specific configuration of the KasperskyOS kernel. If this flag is available, the kernel configuration also defines the specific actions to take with the hardware platform: shut down or restart. • Flags that determine how to commit virtual memory dynamically allocated by a process. One of the following flags can be specified: ◦ `TaskMemoryLocked` – this flag prohibits committing virtual memory in lazy mode. ◦ `TaskOvercommitWaitFree` – if there is insufficient physical memory and a process thread queries virtual memory that must be committed in lazy mode, this thread is locked while waiting for the necessary amount of physical memory to be freed. ◦ `TaskOvercommitException` – if there is insufficient physical memory and a process thread queries virtual memory that must be committed in lazy mode, an exception of the `EXCEPTION_OVERCOMMIT` type is thrown in this thread. The constant `EXCEPTION_OVERCOMMIT` is defined in the header file `sysroot-*-kos/include/hal/exc_codes.h` from the KasperskyOS SDK.

Function	Information about the function
`KnTaskGetGsi()`	Purpose Gets the GSI address for the calling process. Parameters N/A Returned values Pointer to the GSI. The data type for GSI storage is defined in the header file `sysroot-*-kos/include/task/pcbpage.h` from the KasperskyOS SDK.
`KnTaskGetPcb()`	Purpose Gets the address of the process control block (PCB) for the calling process. Parameters N/A Returned values Pointer to the PCB. The data type for PCB storage is defined in the header file `sysroot-*-kos/include/task/pcbpage.h` from the KasperskyOS SDK.
`KnTaskGetEnv()`	Purpose Gets the SCP address of the calling process. Parameters • [out] `envSize` – pointer to the size of the SCP (in bytes). Returned values Pointer to the SCP, or `RTL_NULL` if the process does not have an SCP.
`KnTaskSetEnv()`	Purpose Writes data to the SCP of a process. Parameters • [in] `task` – process handle. • [in] `env` – pointer to the buffer containing data to be written to the SCP. • [in] `envSize` – size (in bytes) of the data to be written to the SCP. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnTaskFreeEnv()`	Purpose Deletes the SCP of the calling process. Parameters N/A Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KnTaskResume()`	Purpose Resumes a process. Parameters • [in] `task` – process handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnTaskExit()`	Purpose Terminates the calling process. Parameters • [in] `status` – exit code of the process. Returned values Error code. Additional information Does not terminate the process as long as it contains threads attached to interrupts.
`KnTaskTerminate()`	Purpose Terminates a process. Parameters • [in] `task` – process handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnTaskGetExitStatus()`	Purpose Gets information about the reason for process termination. Parameters • [in] `task` – handle of the terminated process. • [out] `status` – pointer to the value indicating the reason for process termination. The data type for storing this value is defined in the header file `sysroot-*-kos/include/task/pcbpage.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information You can use the `status` parameters to get the following values: • `TaskExitUnexpected` – the process terminated unexpectedly. • `TaskExitNormal` – the process terminated on its own initiative. • `TaskExitTerminated` – the process was terminated by an external request.

Function	Information about the function
`KnTaskGetExceptionInfo()`	Purpose Gets information about an unhandled exception that led to an unexpected termination of a process. Parameters • [in] `task` – handle of the unexpectedly terminated process. • [out] `excType` – pointer to an exception code that is non-specific for any processor architecture. The data type for storing this code is defined in the header file `sysroot-*-kos/include/hal/exc_codes.h` from the KasperskyOS SDK. • [out] `excNo` – pointer to an exception code that is specific to the utilized processor architecture. The data type for storing this code is defined in the header file `sysroot-*-kos/include/hal/exc_codes.h` from the KasperskyOS SDK. • [out] `exceptionContext` – pointer to the structure containing the context of the thread in which the exception occurred. The type of structure is defined in the header file `sysroot-*-kos/include/hal/*/frame.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnTaskGetThreadContext()`	Purpose Gets the context of a thread that is part of a frozen process. Parameters • [in] `task` – handle of the frozen process. • [in] `index` – thread index. It is used to enumerate threads. Enumeration starts with zero. A thread in which an unhandled exception occurred has a zero index. • [out] `context` – pointer to the structure containing the thread ID (TID) and context of the thread. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnTaskGetNextVmRegion()`	Purpose Gets information about the virtual memory region that belongs to a frozen process. Parameters • [in] `task` – handle of the frozen process. • [in,optional] `after` – address after which the virtual memory region was placed, or `RTL_NULL` if you need to receive information about the virtual memory region that was placed first. • [out] `next` – pointer to the base address of the virtual memory region. • [out] `size` – pointer to the size of the virtual memory region (in bytes). • [out] `flags` – pointer to the flags indicating the parameters of the virtual memory region. The flags are defined in the header file `sysroot-*-kos/include/vmm/flags.h` from the KasperskyOS SDK. • [out,optional] `outHandle` – pointer to the handle of the MDL buffer mapped to the virtual memory region. Creates an MDL buffer from physical memory mapped to a virtual memory region (even if there is already an MDL buffer mapped to this virtual memory region). If you specify `RTL_NULL`, an MDL buffer will not be created. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KnTaskTerminateAfterFreezing()`	Purpose Terminates a frozen process. Parameters • [in] `task` – handle of the frozen process. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnTaskGetExitCode()`	Purpose Gets the exit code of a process that terminated on its own initiative. Parameters • [in] `task` – handle of the terminated process. • [out] `exitCode` – pointer to the process exit code. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnTaskGetId()`	Purpose Gets the process ID (PID) for the calling process. Parameters N/A Returned values Process ID. The ID type is defined in the header file `sysroot-*-kos/include/task/pidtype.h` from the KasperskyOS SDK.
`KnTaskGetName()`	Purpose Gets the name of a calling process. Parameters • [out] `name` – pointer to the buffer used to store the process name. • [in] `msize` – size of the buffer used to store the process name (in bytes). Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnTaskGetPath()`	Purpose Gets the path to the executable file from which the calling process was created. Parameters • [out] `path` – pointer to the buffer used to store the path to the executable file. If the path to the executable file was not specified when the process was created, an empty string will be saved in the buffer. • [in] `msize` – size (in bytes) of the buffer used to store the path to the executable file. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KnTaskGetInitialThreadPriority()`	Purpose Gets the priority of the initial thread of a process. Parameters • [in] `task` – process handle. • [out] `priority` – pointer to the priority of the initial thread of the process. The data type for storing the priority of a thread is defined in the header file `sysroot-*-kos/include/coresrv/thread/thread_api.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnTaskSetInitialThreadPriority()`	Purpose Sets the priority of the initial thread of a process. Parameters • [in] `task` – process handle. • [in] `priority` – priority of the initial thread of a process. The parameter type is defined in the header file `sysroot-*-kos/include/coresrv/thread/thread_api.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnTaskSetExceptionHandler()`	Purpose Registers the exception handling function for the calling process. Parameters • [in,optional] `handler` – ID of the exception handling function, or `RTL_NULL` to deregister the previously registered function without registering a new one. The parameter type is defined in the header file `sysroot-*-kos/include/task/pcbpage.h` from the KasperskyOS SDK. Returned values: ID of the previously registered exception handling function if one exists, otherwise `RTL_NULL`. The type of returned value is defined in the header file `sysroot-*-kos/include/task/pcbpage.h` from the KasperskyOS SDK.
`KnTaskLoadSeg()`	Purpose Loads an ELF image segment into process memory. Parameters • [in] `h` – process handle. • [in] `seg` – pointer to the structure describing the loaded ELF image segment. If the `loadAddr` field of this structure is set equal to `0`, the segment load address will be selected automatically. The type of structure is defined in the header file `sysroot-*-kos/include/coresrv/vmm/vmm_types.h` from the KasperskyOS SDK. • [out] `actual` – pointer to the load address of the ELF image segment. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KnTaskVmReserve()`	Purpose Reserves a virtual memory region in a process. Parameters • [in] `task` – process handle. • [in,optional] `hint` – page-aligned, preferred base address of the virtual memory region, or `RTL_NULL` to select this address automatically. • [in] `size` – size (in bytes) of the virtual memory region. • [in] `flags` – flags defining the parameters of the virtual memory region. The flags are defined in the header file `sysroot-*-kos/include/vmm/flags.h` from the KasperskyOS SDK. • [out] `addr` – pointer to the base address of the reserved virtual memory region. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information In the `flags` parameter, you can specify the following flags: • `VMM_FLAG_LOCKED` – commits the virtual memory region while allocating the entire required volume of physical memory. If committing virtual memory in lazy mode is prohibited at the system level or in the process, either the reserved region will be committed with the entire amount of necessary physical memory allocated or the function will end with an error if there is insufficient physical memory, even if the `VMM_FLAG_LOCKED` flag is not specified. If committing virtual memory in lazy mode is not prohibited, and the `VMM_FLAG_LOCKED` flag is not specified, the reserved region will be committed in lazy mode. The function does not commit the reserved region of virtual memory but instead only determines how the memory will be committed during subsequent actions with this region. • `VMM_FLAG_READ`, `VMM_FLAG_WRITE`, `VMM_FLAG_EXECUTE` and `VMM_FLAG_RWX_MASK` – flags defining the access rights to the virtual memory region. • `VMM_FLAG_LOW_GUARD`, `VMM_FLAG_HIGH_GUARD` – adds a guard page to the beginning and end of the virtual memory region, respectively. The size of the guard page is not included in the size of the virtual memory region. Permissible combinations of flags defining the access rights to the virtual memory region: • `VMM_FLAG_READ` – read access. • `VMM_FLAG_READ | VMM_FLAG_WRITE` – read-and-write access. • `VMM_FLAG_READ | VMM_FLAG_EXECUTE` – read-and-execute access. • `VMM_FLAG_RWX_MASK` or `VMM_FLAG_READ | VMM_FLAG_WRITE | VMM_FLAG_EXECUTE` – read/write/execute access. • `0` – access denied.

Function	Information about the function
`KnTaskVmReserveForElf()`	Purpose Reserves a virtual memory region in a process for loading the ELF image. Parameters • [in] `task` – process handle. • [in] `elfDescr` – pointer to the object describing the ELF image. The data type for storing the object is defined in the header file `sysroot-*-kos/include/coresrv/elf/elf_api.h` from the KasperskyOS SDK. • [out] `relocBase` – pointer to the ELF image load offset (in bytes). Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information If committing virtual memory in lazy mode is prohibited at the system level (the kernel startup parameter `vmm.overcommit=disabled` is defined) or in the process (the `KOS_TASK_FLAG_LOCK_MEM` flag was specified when creating a process using the task.h API), either the reserved region will be committed with the entire amount of necessary physical memory allocated or the function will end with an error if there is insufficient physical memory. (For more information about committing virtual memory, see "Allocating and freeing memory".) If committing virtual memory in lazy mode is not prohibited, the reserved region will be committed in lazy mode. The function does not commit the reserved region of virtual memory but instead only determines how the memory will be committed during subsequent actions with this region.
`KnTaskVmFree()`	Purpose Frees the virtual memory region that was reserved by calling the `KnTaskVmReserve()` function. Parameters • [in] `task` – process handle. • [in] `base` – page-aligned base address of the virtual memory region. • [in] `size` – size (in bytes) of the virtual memory region. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnTaskSetInitialState()`	Purpose Defines the program entry point and the ELF image load offset. Parameters • [in] `h` – process handle. • [in] `state` – pointer to the structure containing the address of the program entry point and the ELF image load offset (in bytes). Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KnTaskLoadElfSyms()`	Purpose Loads the symbol table `.symtab` and string table `.strtab` into process memory. Parameters • [in] `h` – process handle. • [in] `symTabSeg` – pointer to the structure describing the loaded ELF image segment with the symbol table `.symtab`. The type of structure is defined in the header file `sysroot-*-kos/include/coresrv/vmm/vmm_types.h` from the KasperskyOS SDK. • [in] `symTabSize` – size of the symbol table `.symtab` (in bytes). • [in] `strTabSeg` – pointer to the structure describing the loaded ELF image segment with the string table `.strtab`. The type of structure is defined in the header file `sysroot-*-kos/include/coresrv/vmm/vmm_types.h` from the KasperskyOS SDK. • [in] `strTabSize` – size of the string table `.strtab` (in bytes). Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnTaskSetInitialPolicy()`	Purpose Defines the scheduler class and priority of the initial thread of a process. Parameters • [in] `task` – process handle. • [in] `policy` – scheduler class of the initial thread of the process. The parameter type is defined in the header file `sysroot-*-kos/include/thread/tidtype.h` from the KasperskyOS SDK. • [in] `priority` – priority of the initial thread of a process. The parameter type is defined in the header file `sysroot-*-kos/include/coresrv/thread/thread_api.h` from the KasperskyOS SDK. • [in,optional] `param` – pointer to the parameters of the initial thread scheduler class, or `RTL_NULL`. The data type for storing these parameters is defined in the header file `sysroot-*-kos/include/thread/tidtype.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information In the `param` parameter, you can specify `RTL_NULL` if you need to define the default parameters for the scheduler class of the initial thread of the process. In the `param` parameter, you must specify `RTL_NULL` if the scheduler class of the initial thread of the process does not have any parameters.
`KnTaskReseedAslr()`	Purpose Defines the seed value for ASLR support in the defined process. Parameters • [in] `task` – process handle. • [in] `seed` – pointer to the buffer containing the seed value of the random number generator. • [in] `seedSize` – size (in bytes) of the buffer containing the seed value of the random number generator. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KnTaskGetElfSyms()`	Purpose Gets the addresses and sizes of the symbol table `.symtab` and string table `.strtab` for the calling process. Parameters • [out] `relocBase` – pointer to the ELF image load offset (in bytes). Add the ELF image load offset to the address of a symbol from the symbol table `.symtab` to get the address of this symbol in process memory. • [out] `syms` – pointer to the address of the symbol table `.symtab`. • [out] `symsCnt` – pointer to the size of the symbol table `.symtab` (in bytes). • [out] `strs` – pointer to the address of the string table `.strtab`. • [out] `strsCnt` – pointer to the size of the string table `.strtab` (in bytes). Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information If the symbol table `.symtab` and string table `.strtab` are not loaded into process memory by calling the `KnTaskLoadElfSyms()` function, the `KnTaskGetElfSyms()` function returns `rcOk` (if there are no other errors). In this case, the received addresses of the tables have `RTL_NULL` values, and the received sizes of the tables are zero.
`KnTaskGetIdByHandle()`	Purpose Gets the process ID (PID). Parameters • [in] `task` – process handle. • [out] `taskId` – pointer to the process ID. The ID type is defined in the header file `sysroot-*-kos/include/task/pidtype.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnTaskPanic()`	Purpose Initiates an exception that cannot be handled and leads to the process terminating. Parameters N/A Returned values N/A

Function	Information about the function
`KnTaskTransferResource()`	Purpose Transfers a handle to a process that is not yet running. Parameters • [in] `task` – process handle. • [in] `srcHandle` – transferred handle. • [in] `srcBadge` – handle of the resource transfer context object. • [in] `dstRights` – permissions mask of the descendant of the transferred handle. • [out] `outDstHandle` – pointer to the descendant of the transferred handle (from the handle space of the process that received the handle). Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Initializing IPC transport for interprocess communication and managing IPC request processing (transport-kos.h, transport-kos-dispatch.h)

APIs are defined in the header files `transport-kos.h` and `transport-kos-dispatch.h` from the KasperskyOS SDK that are located at the path `sysroot-*-kos/include/coresrv/nk`.

API capabilities:

- Initialize IPC transport on the client side and on the server side.

  IPC transport is a wrapper for system calls for sending and receiving IPC messages and works separately with the constant part and arena of IPC messages. IPC transport is used by the transport code on the client side and by the code for receiving and sending IPC messages on the server side.

- Start the loop for processing IPC requests on the server side.

- Copy data to the IPC message arena.

Information about API functions is provided in the tables below.

This section contains API usage examples. These examples use dynamically and statically created IPC channels, and programs acting as servers have the following formal specification:

FsDriver.edl

```
task class FsDriver
components {
    operationsComp : Operations
}
```

Operations.cdl

```
component Operations
endpoints {
    fileOperations : FileIface
}
```

FileIface.idl

```
package FileIface
interface {
    Open(in array<UInt8, 1024> path);
    Read(out sequence<UInt8, 2048> content);
}
```

Initializing IPC transport for interprocess communication

To initialize IPC transport for interaction with other processes, call the `NkKosTransport_Init()` or `NkKosTransportSync_Init()` function declared in the header file `transport-kos.h`.

Example use of the `NkKosTransport_Init()` function on the client side:

```
int main(int argc, const char *argv[])
{
    /* Declare the structure containing the IPC transport parameters */
    NkKosTransport driver_transport;
    /* Declare the proxy object. (The type of proxy object is automatically
     * generated transport code.) */
    struct FileIface_proxy file_operations_proxy;
    /* Declare the structures for saving the constant part of an IPC request and
     * IPC response for the endpoint method. (The types of structures are automatically
     * generated transport code.) */
    struct FileIface_Open_req req;
    struct FileIface_Open_res res;
    ...
    Retcode rc = rcOk;
    Handle selfHandle = INVALID_HANDLE;
    Handle serverHandle = INVALID_HANDLE;
    Handle callableHandle = INVALID_HANDLE;
    DcmHandle subscrHandle = DCM_INVALID_HANDLE;
    DcmHandle connectHandle = DCM_INVALID_HANDLE;
    KosTask *self = RTL_NULL;
    nk_handle_desc_t desc = nk_handle_desc();
    char nodeName[UCORE_STRING_SIZE];
    DcmPublicationStatus status;

    /* Create a connection to DCM */
    rc = DcmInit();
    if (rc != rcOk)
    {
        return rc;
    }

   /* Get the handle of the client process */
    self = KosTaskGetSelf();
    if (self != RTL_NULL)
    {
        selfHandle = KosTaskGetHandle(self);
        KosTaskPut(self);
    }

/ * Create a DCM handle that lets you receive
     * notifications about the publication and unpublication of endpoints */
    rc = DcmSubscribeToEndpoints(
        "FileIface",
        RTL_NULL,
        RTL_NULL,
        INVALID_HANDLE,
        selfHandle,
```

```
        &subscrHandle);
    if (rc != rcOk)
    {
        return rc;
    }

    /* Extract a notification about endpoint publication */
    rc = DcmReadPubQueue(
        subscrHandle,
        TIMEOUT_MSEC,
        &serverHandle,
        RTL_NULL,
        0,
        nodeName,
        sizeof(nodeName),
        &status);
    if ((rc != rcOk) || (status != DcmNodePublished))
    {
        DcmCloseHandle(subscrHandle);
        return rc;
    }

    /* Create a DCM handle that lets you execute a request to create
     * an IPC channel with a server */
    rc = DcmCreateConnection(
        "FileIface", nodeName, serverHandle, selfHandle, &connectHandle);
    if (rc != rcOk)
    {
        DcmCloseHandle(subscrHandle);
        KnHandleClose(serverHandle);
        return rc;
    }

    /* Fulfill a request to create an IPC channel with a server */
    rc = DcmConnectToEndpoint(connectHandle, TIMEOUT_MSEC, &desc);
    if (rc != rcOk)
    {
        DcmCloseHandle(subscrHandle);
        DcmCloseHandle(connectHandle);
        KnHandleClose(serverHandle);
        return rc;
    }

    DcmCloseHandle(subscrHandle);
    DcmCloseHandle(connectHandle);
    KnHandleClose(serverHandle);

    /* Delete a connection to DCM */
    DcmFini();

    /* Get a callable handle */
    callableHandle = nk_get_handle(&desc);

    /* Initialize the structure containing the IPC transport parameters */
    NkKosTransport_Init(&driver_transport, callableHandle, NK_NULL, 0);

    /* Initialize the proxy object. (The proxy object initialization method is
     * is automatically generated transport code.) */
    FileIface_proxy_init(
        &file_operations_proxy,
        &driver_transport.base,
        INVALID_RIID);
    ...
    /* Call the endpoint method. (The method is automatically
     * generated transport code.) */
    strncpy(req.path, "/example/file/path", sizeof(req.path));
    if (FileIface_Open(file_operations_proxy.base, &req, RTL_NULL,
                       &res, RTL_NULL) != NK_EOK) {
        ...
    }
    ...
}
```

If a client needs to use several endpoints, the same number of proxy objects must be initialized. When initializing each proxy object, you need to specify the IPC transport that is associated through the client IPC handle with the relevant server. When initializing multiple proxy objects pertaining to the endpoints of one server, you can specify the same IPC transport that is associated with this server.

Example use of the `NkKosTransport_Init()` function on the server side:

```
int main(int argc, const char *argv[])
{
    /* Declare the structure containing the IPC transport parameters */
    NkKosTransport transport;
    ...
    Retcode rc = rcOk;
    Handle listenerHandle = INVALID_HANDLE;
    Handle callableHandle = INVALID_HANDLE;
    Handle selfHandle = INVALID_HANDLE;
    DcmHandle pubHandle = DCM_INVALID_HANDLE;
    DcmHandle connectHandle = DCM_INVALID_HANDLE;
    KosTask *self = RTL_NULL;
    char clientName[UCORE_STRING_SIZE];

    /* Create a connection to DCM */
    rc = DcmInit();
    if (rc != rcOk)
    {
        return rc;
    }

    /* Get the handle of the server process */
    self = KosTaskGetSelf();
    if (self != RTL_NULL)
    {
        selfHandle = KosTaskGetHandle(self);
        KosTaskPut(self);
    }

    /* Publish the endpoint to DCM */
    rc = DcmPublishEndpoint(
        "FileIface",
        "operationsComp.fileOperations",
        selfHandle,
        &pubHandle);
    if (rc != rcOk)
    {
        return rc;
    }

    /* Create a listener handle */
    rc = KnHandleCreateListener(&listenerHandle);
    if (rc != rcOk)
    {
        DcmCloseHandle(pubHandle);
        return rc;
    }

    /* Create a callable handle. (Endpoint ID
     * FsDriver_operationsComp_fileOperations_iid is
     * automatically generated transport code.) */
    rc = KnHandleCreateUserObjectEx(
        HANDLE_TYPE_USER_FIRST,
        OCAP_IPC_CALL | OCAP_HANDLE_TRANSFER,
        RTL_NULL,
        listenerHandle,
        FsDriver_operationsComp_fileOperations_iid,
        &callableHandle);
    if (rc != rcOk)
    {
        DcmCloseHandle(pubHandle);
        KnHandleClose(listenerHandle);
        return rc;
    }

    /* Receive a client request to create an IPC channel */
    rc = DcmListen(
        pubHandle,
        INFINITE_TIMEOUT,
        &connectHandle,
        RTL_NULL,
        clientName,
        sizeof(clientName));
    if (rc != rcOk)
    {
        KnHandleClose(callableHandle);
        KnHandleClose(listenerHandle);
        DcmCloseHandle(pubHandle);
```

```
        return rc;
    }

    /* Accept a client request to create an IPC channel */
    rc = DcmAcceptEndpointConnection(
        connectHandle,
        nk_handle_desc(callableHandle, NK_SAME));
    if (rc != rcOk)
    {
        KnHandleClose(callableHandle);
        KnHandleClose(listenerHandle);
        DcmCloseHandle(pubHandle);
        DcmCloseHandle(connectHandle);
        return rc;
    }

    KnHandleClose(callableHandle);
    DcmCloseHandle(pubHandle);
    DcmCloseHandle(connectHandle);

    /* Delete a connection to DCM */
    DcmFini();

    /* Initialize the structure containing the IPC transport parameters */
    NkKosTransport_Init(&transport, listenerHandle, NK_NULL, 0);
    ...
    /* IPC request processing loop */
    do
    {
        ...
        /* Get the IPC request */
        rc = nk_transport_recv(&transport.base, ...);
        if (rc == NK_EOK) {
            /* Process the IPC request by calling the dispatcher. (The dispatcher
             * is automatically generated transport
             * code.) */
            rc = FsDriver_entity_dispatch(...);
            if (rc == NK_EOK) {
                /* Send an IPC response */
                rc = nk_transport_reply(&transport.base, ...);
            }
        }
    }
    while (rc == NK_EOK)

    KnHandleClose(listenerHandle);

    return EXIT_SUCCESS;
}
```

If a server processes IPC requests received through multiple IPC channels, the following special considerations should be taken into account:

- If a listener handle is associated with all IPC channels, IPC interaction with all clients can use the same IPC transport associated with this listener handle.

- If IPC channels are associated with different listener handles, IPC interaction with each group of clients corresponding to the same listener handle must use a separate IPC transport associated with this listener handle. In this case, IPC requests can be processed in parallel threads if you are using a thread-safe implementation of endpoint methods.

The `NkKosTransportSync_Init()` function initializes IPC transport with support for interrupting the `Call()` and `Recv()` locking system calls. (For example, an interrupt of these calls may be required for correct termination of the process that is executing them.) To interrupt the `Call()` and `Recv()` system calls, use the API ipc_api.h.

The `NkKosSetTransportTimeouts()` function declared in the header file `transport-kos.h` defines the maximum blocking duration for `Call()` and `Recv()` system calls used for IPC transport.

Starting the IPC request processing loop

The IPC request processing loop on a server includes the following steps:

1. Receive an IPC request.
2. Process the IPC request.
3. Send an IPC response.

Each step of this loop can be completed separately by sequentially calling the `nk_transport_recv()`, dispatcher, and `nk_transport_reply()` functions. (The `nk_transport_recv()` and `nk_transport_reply()` functions are declared in the header file `sysroot-*-kos/include/nk/transport.h` from the KasperskyOS SDK.) You can also call the `NkKosTransport_Dispatch()`, `NkKosDoDispatch()` or `NkKosDoDispatchEx()` function in which this loop is completed in its entirety. (The `NkKosTransport_Dispatch()` function is declared in the header file `transport-kos.h`, and the `NkKosDoDispatch()` and `NkKosDoDispatchEx()` functions are declared in the header file `transport-kos-dispatch.h`.) Use of the `NkKosDoDispatch()` or `NkKosDoDispatchEx()` function is more convenient than the `NkKosTransport_Dispatch()` function because they require fewer preparatory operations (for example, you do not need to initialize IPC transport).

You can initialize the structure passed to the `NkKosDoDispatch()` or `NkKosDoDispatchEx()` function via the `info` parameter by using the macros defined in the header file `transport-kos-dispatch.h`. You can use the `syncHandle` parameter of the `NkKosDoDispatchEx()` function to interrupt the system call `Recv()` for the purpose of exiting the IPC request processing loop. You can use the `cb` parameter of the `NkKosDoDispatchEx()` function to define the callback functions that are called after completing the `Recv()` and `Reply()` system calls, and after exiting the IPC request processing loop (you do not need to define all functions). The callback function corresponding to the `enterDispatchProcessing` pointer is called immediately after the `Recv()` system call is completed. The callback function corresponding to the `leaveDispatchProcessing` pointer is called immediately after the `Reply()` system call is completed. The callback function corresponding to the `stopDispatchLoop` pointer is called immediately after exiting the IPC request processing loop. The `callbackContext` pointer is passed to each of these callback functions. For example, these callback functions are used to implement the thread pool for processing IPC requests.

The `NkKosTransport_Dispatch()`, `NkKosDoDispatch()`, and `NkKosDoDispatchEx()` functions can be called from parallel threads if you are using a thread-safe implementation of endpoint methods.

Example use of the `NkKosDoDispatch()` function:

```
/* This function implements the endpoint method. */
static nk_err_t Open_impl(...)
{
    ...
}

/* This function implements the endpoint method. */
static nk_err_t Read_impl(...)
{
    ...
}

/* This function initializes the pointers to functions implementing the endpoint methods.
* (These pointers are used by the dispatcher to call functions implementing the
* endpoint methods.The types of structures are automatically generated
* transport code.) */
static struct FileIface *CreateFileOperations()
{
    static const struct FileIface_ops ops = {
        .Open = Open_impl,
        .Read = Read_impl
```

```
    };
    static struct FileIface impl = {
        .ops = &ops
    };
    return &impl;
}

int main(int argc, const char *argv[])
{
    /* Declare the stubs. (The types of stubs are automatically generated
     * transport code. */
    struct Operations_component component;
    struct FsDriver_entity entity;
    ...
    /* Get the listener handle */
    Handle handle = SLStaticResolveHandle("driver_connection", HandleTypeServer);
    assert(handle != INVALID_HANDLE);

    /* Initialize the stubs. (Methods for initializing stubs are
     * automatically generated transport code.) */Function
     * CreateFileOperations() is implemented by the developer of the
     * KasperskyOS-based solution to initialize
     * pointers to functions implementing the endpoint methods.) */
    Operations_component_init(&component, CreateFileOperations());
    FsDriver_entity_init(&entity, &component);

    /* Initialize the structure that is required for the
     * NkKosDoDispatch() function to use transport code. */
    NkKosDispatchInfo info = NK_TASK_DISPATCH_INFO_INITIALIZER(FsDriver, entity);

    /* Start the IPC request processing loop */
    NkKosDoDispatch(handle, &info);

    return EXIT_SUCCESS;
}
```

Example use of the `NkKosTransport_Dispatch()` function:

```
/* This function implements the endpoint method. */
static nk_err_t Open_impl(...)
{
    ...
}

/* This function implements the endpoint method. */
static nk_err_t Read_impl(...)
{
    ...
}

/* This function initializes the pointers to functions implementing the endpoint methods.
* (These pointers are used by the dispatcher to call functions implementing the
* endpoint methods.The types of structures are automatically generated
* transport code.) */
static struct FileIface *CreateFileOperations()
{
    static const struct FileIface_ops ops = {
        .Open = Open_impl,
        .Read = Read_impl
    };
    static struct FileIface impl = {
        .ops = &ops
    };
    return &impl;
}

int main(int argc, const char *argv[])
{
    /* Declare the structure containing the IPC transport parameters */
    NkKosTransport transport;
    /* Declare the stubs. (The types of stubs are automatically generated
     * transport code. */
    struct Operations_component component;
    struct FsDriver_entity entity;
    /* Declare the unions of the constant part of IPC requests and
     * IPC responses. (Types of unions are automatically generated
     * transport code.) */
    union FsDriver_entity_req req;
    union FsDriver_entity_res res;
    /* Declare the array for the IPC response arena. (The array size is
```

```
     * automatically generated transport code.) */
    char res_buffer[FsDriver_entity_res_arena_size];
    /* Declare and initialize the arena descriptor of the IPC response.
     * (The type of handle and its initialization macro are defined in the header file
     * sysroot-*-kos/include/nk/arena.h from the KasperskyOS SDK.) */
    struct nk_arena res_arena = NK_ARENA_INITIALIZER(res_buffer,
                                                     res_buffer + sizeof(res_buffer));
    ...
    /* Get the listener handle */
    Handle handle = SLStaticResolveHandle("driver_connection", HandleTypeServer);
    assert(handle != INVALID_HANDLE);

    /* Initialize the structure containing the IPC transport parameters */
    NkKosTransport_Init(&transport, handle, NK_NULL, 0);

    /* Initialize the stubs. (Methods for initializing stubs are
     * automatically generated transport code.) */Function
     * CreateFileOperations() is implemented by the developer of the
     * KasperskyOS-based solution to initialize
     * pointers to functions implementing the endpoint methods.) */
    Operations_component_init(&component, CreateFileOperations());
    FsDriver_entity_init(&entity, &component);

    /* Start the IPC request processing loop. (The dispatcher FsDriver_entity_dispatch
     * is automatically generated transport code.) */
    NkKosTransport_Dispatch(&transport.base, FsDriver_entity_dispatch,
                            &entity, &req, sizeof(FsDriver_entity_req),
                            RTL_NULL, &res, &res_arena);

    return EXIT_SUCCESS;
}
```

Copying data to the IPC message arena

To copy a string to the IPC message arena, call the `NkKosCopyStringToArena()` function declared in the header file `transport-kos.h`. This function reserves a segment of the arena and copies a string to this segment.

Example use of the `NkKosCopyStringToArena()` function:

```
static nk_err_t Read_impl(struct FileIface *self,
                          const struct FileIface_Read_req *req,
                          const struct nk_arena* req_arena,
                          struct FileIface_Read_res* res,
                          struct nk_arena* res_arena)
{
    /* Copy the string to the IPC response arena */
    if (NkKosCopyStringToArena(res_arena, &res->content,
                               "CONTENT OF THE FILE") != rcOk) {
        ...
    }

    return NK_EOK;
}
```

Information about API functions

transport-kos.h functions

Function	Information about the function
`NkKosTransport_Init()`	Purpose Initializes IPC transport. Parameters • [out] `transport` – pointer to the structure containing the IPC transport parameters. • [in] `handle` – client or server IPC handle. • [in] `view` – parameter that must have the value `NK_NULL`. • [in] `size` – parameter that must have the value `0`. Returned values N/A
`NkKosTransportSync_Init()`	Purpose Initializes IPC transport with support for interrupting the `Call()` and/or `Recv()` system calls. Parameters • [out] `transport` – pointer to the structure containing the IPC transport parameters. • [in] `handle` – client or server IPC handle. • [in,optional] `callSyncHandle` – handle of the IPC synchronization object for `Call()` system calls, or `INVALID_HANDLE` if an interrupt of `Call()` system calls is not required. • [in,optional] `recvSyncHandle` – handle of the IPC synchronization object for `Recv()` system calls, or `INVALID_HANDLE` if an interrupt of `Recv()` system calls is not required. Returned values N/A
`NkKosSetTransportTimeouts()`	Purpose Defines the maximum blocking duration for `Call()` and `Recv()` system calls used for IPC transport. Parameters • [out] `transport` – pointer to the structure containing the IPC transport parameters. • [in] `recvTimeout` – maximum blocking duration for `Recv()` system calls (in milliseconds), or `INFINITE_TIMEOUT` to define an unlimited blocking duration. • [in] `callTimeout` – maximum blocking duration for `Call()` system calls (in milliseconds), or `INFINITE_TIMEOUT` to define an unlimited blocking duration. Returned values N/A

Function	Information about the function
`NkKosTransport_Dispatch()`	Purpose Starts the IPC request processing loop. Parameters • [in] `transport` – pointer to the `base` field of the structure containing the IPC transport parameters. • [in] `dispatch` – pointer to the dispatcher (dispatch method) from the transport code. The dispatcher is named `<process class name>_entity_dispatch`. • [in] `impl` – pointer to the stub, which consists of a structure with the type `<process class name>_entity` from the transport code. The dispatcher uses this structure to get the pointers to functions implementing endpoint methods. • [out] `req` – pointer to the union with the type `<process class name>_entity_req` from the transport code. This union is intended for storing the constant part of IPC requests for any methods of endpoints provided by the server. • [in] `req_size` – maximum size (in bytes) of the constant part of IPC requests. It is defined as `sizeof(<process class name>_entity_req)`, where `<process class name>_entity_req` is the type from the transport code. • [in,out,optional] `req_arena` – pointer to the IPC request arena descriptor, or `RTL_NULL` if an IPC request arena is not in use. The type of handle is defined in the header file `sysroot-*-kos/include/nk/arena.h` from the KasperskyOS SDK. • [out] `res` – pointer to the union with the type `<process class name>_entity_res` from the transport code. This union is intended for storing the constant part of IPC responses for any methods of endpoints provided by the server. • [in,out,optional] `res_arena` – pointer to the IPC response arena descriptor, or `RTL_NULL` if an IPC response arena is not in use. The type of handle is defined in the header file `sysroot-*-kos/include/nk/arena.h` from the KasperskyOS SDK. Returned values If unsuccessful, it returns an error code.
`NkKosCopyStringToArena()`	Purpose Reserves a segment of the arena and copies a string to this segment. Parameters • [in,out] `arena` – pointer to the arena descriptor. The type of handle is defined in the header file `sysroot-*-kos/include/nk/arena.h` from the KasperskyOS SDK. • [out] `field` – pointer to the arena chunk descriptor where the string is copied. The type of handle is defined in the header file `sysroot-*-kos/include/nk/types.h`. • [in] `src` – pointer to the string to be copied to the IPC message arena. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

transport-kos-dispatch.h functions

Function	Information about the function
`NkKosDoDispatch()`	Purpose Starts the IPC request processing loop. Parameters • [in] `h` – server IPC handle. • [in] `info` – pointer to the structure containing the data required by the function to use transport code (including the pointers to the dispatcher and stub, sizes of the constant part and IPC message arena). Returned values N/A
`NkKosDoDispatchEx()`	Purpose Starts the IPC request processing loop. Parameters • [in] `h` – server IPC handle. • [in] `info` – pointer to the structure containing the data required by the function to use transport code (including the pointers to the dispatcher and stub, sizes of the constant part and IPC message arena). • [in,optional] `cb` – pointer to the structure containing pointers to the callback functions that are called after completing the `Recv()` and `Reply()` system calls, and after exiting the IPC request processing loop. In addition to the pointers to callback functions, this structure contains the pointer that is passed to these callback functions when they are called. You can set this parameter to `RTL_NULL` if you do not need to define callback functions. • [in,optional] `syncHandle` – handle of the IPC synchronization object, or `INVALID_HANDLE` if an interrupt of the `Recv()` system call is not required. Returned values N/A

Initializing and de-initializing IPC transport for querying the security module (transport-kos-security.h)

The API is defined in the header file `sysroot-*-kos/include/coresrv/nk/transport-kos-security.h` from the KasperskyOS SDK.

The API initializes and de-initializes IPC transport for querying the Kaspersky Security Module via the security interface. Transport code works on top of IPC transport.

Information about API functions is provided in the table below.

This section provides an example of initializing IPC transport to query the security module. In this example, the program that queries the security module has the following formal specification:

Verifier.edl

```
task class Verifier
security Approve
```

Approve.idl

```
package Approve
interface {
    Check(in UInt32 port);
}
```

Fragment of the policy description in the example:

security.psl

```
...
security src=Verifier, method=Check { assert (message.port > 80) }
...
```

Using the API

To initialize IPC transport for querying the security module, call the `NkKosSecurityTransport_Init()` function.

Example use of the `NkKosSecurityTransport_Init()` function:

```
int main(void)
{
    /* Declare the structure containing the IPC transport parameters for querying the
     * security module */
    NkKosSecurityTransport security_transport;
    /* Declare the proxy object. (The type of proxy object is automatically
     * generated transport code.) */
    struct Approve_proxy security_proxy;
    /* Declare the structures for saving the constant part of an IPC request and IPC response for the
     * security interface method. (The types of structures are automatically generated
     * transport code.) */
    struct Approve_Check_req security_req;
    struct Approve_Check_res security_res;
    /* Initialize the structure containing the IPC transport parameters for querying the
     * security module */
    if (NkKosSecurityTransport_Init(&security_transport, NK_NULL, 0) == NK_EOK) {
        /* Initialize the proxy object. (The proxy object initialization method and the
         * security interface ID Verifier_securityIid are
         * automatically generated transport code.) */
        Approve_proxy_init(&security_proxy, &security_transport.base, Verifier_securityIid);
    }
...
    /* Call the security interface method. (The method is automatically generated
     * transport code.The method does not pass any data through the security_res parameter.
     * This parameter should be specified only if required by the method implementation.) */
    security_req.port = 80;
    nk_err_t result = Approve_Check(&security_proxy.base, &security_req,
                      NULL, &security_res, NULL);
    if (result == NK_EOK)
        fprintf(stderr, "Granted");
    if (result == NK_EPERM)
        fprintf(stderr, "Denied");
    else
        fprintf(stderr, "Error");
    return EXIT_SUCCESS;
}
```

If a process needs to use several security interfaces, the same number of proxy objects must be initialized by specifying the same IPC transport and the unique IDs of the security interfaces.

An IPC transport for querying the security module that is no longer required must be deinitalized by calling the `NkKosSecurityTransport_Deinit()` function.

Information about API functions

transport-kos-security.h functions

Function	Information about the function
`NkKosSecurityTransport_Init()`	Purpose Initializes IPC transport for querying the Kaspersky Security Module through the security interface. Parameters • [out] `transport` – pointer to the structure containing the IPC transport parameters for querying the security module. • [in] `view` – parameter that must have the value `NK_NULL`. • [in] `size` – parameter that must have the value `0`. Returned values If successful, the function returns `NK_EOK`, otherwise it returns an error code.
`NkKosSecurityTransport_Deinit()`	Purpose De-initializes IPC transport for querying the Kaspersky Security Module through the security interface. Parameters • [in] `transport` – pointer to the structure containing the IPC transport parameters for querying the security module. Returned values N/A

Using a thread pool to process IPC requests (dispatch_pool.h)

The API is defined in the header file `sysroot-*-kos/include/kos/dispatch_pool.h` from the KasperskyOS SDK.

The API lets you create a pool of threads that handle IPC requests on the server side to enhance the productivity of IPC request processing when compared to processing them in a single thread. The threads in the pool process the IPC requests received over the IPC channels associated with the server IPC handle that was defined when the pool was created.

> To use a thread pool to process IPC requests, you must employ a thread-safe implementation of endpoint methods.

Information about API functions is provided in the table below.

Creating a pool

To create a pool, call one of the `KosDispatchPoolStart*()` functions. Functions without the `Ex` postfix create a pool with the default parameters, while functions with the `Ex` postfix let you create a pool with parameters different from the default parameters. Functions with the `Attached` suffix do not return control throughout the entire lifetime of the created pool, while functions with the `Detached` suffix return control immediately after the pool is created. This is because the calling thread manages the pool (creates and terminates threads) in the first case, while a specially created thread manages the pool in the second case.

The fields of the `KosDispatchPoolParams` structure are intended for setting the pool parameters. A pool includes `minThreadCount` threads immediately after the pool is created. As the load on the server increases, the number of threads may increase to `maxThreadCount`. A message about reaching `maxThreadCount` is put into diagnostic output every `exhaustionReportPeriod` ms. Reducing the load on the server may cause the number of threads to decrease to `minThreadCount` because the pool is cleared of idle threads every `shrinkPeriod` ms. No more than `shrinkCount` threads are terminated as a result of each clearing iteration. If the `shrinkPeriod` parameter has the `KOS_DISPATCH_POOL_NO_SHRINK` symbolic constant value, the pool is not cleared of idle threads and the number of threads in a pool that has reached the `maxThreadCount` remains unchanged.

If you specify `RTL_NULL` in the `threadApi` field of the `KosDispatchPoolParams` structure, the `libkos` library will use the default functions (API thread.h) to manage the pool and the pool will consist of standard threads with a priority of 10. (For details about standard threads, see Managing threads (low-level API thread_api.h).) You can replace the default functions with your own if necessary. For example, this may be required if you need to define a standard thread priority higher than 10 or if you need to use the API Pthreads for pool management. To do so, in the `threadApi` field of the `KosDispatchPoolParams` structure, you must specify the pointer to the `KosDispatchPoolThreadApi` structure that contains the ID of the callback function for creating a thread (`create`) and the ID of the callback function that waits for the thread to terminate (`wait`).

The callback function corresponding to the `create` identifier accepts the thread stack size in bytes (the `stackSize` input parameter), the identifier of the function to be executed by the thread (the `routine` input parameter), and the parameters to be passed to the function being executed by the thread (the `arg` input parameter). (The function executed by the thread starts the IPC request processing loop.) The `handle` output parameter is used to pass the identifier of the created thread (for example: handle). If successful, the function returns `rcOk`, otherwise it returns an error code.

The callback function corresponding to the `wait` identifier accepts the thread identifier (the `handle` input parameter) and the time to wait for the thread to terminate (the `timeout` input parameter). It frees up the resources of the terminated thread. Returns `rcOk` if successful. If the thread termination timeout has elapsed, it returns `rcTimeout`. To allow the thread to terminate, the `libkos` library uses the ipc_api.h API to interrupt the `Recv()` system call that is executed by this thread for the purpose of exiting the IPC request processing loop. The maximum time to wait for termination of an idle thread is defined by the `shrinkThreadWaitTimeout` pool parameter (in milliseconds). The maximum time to wait for termination of a thread when deleting a pool is defined by the `stopThreadWaitTimeout` pool parameter (in milliseconds). To set an unlimited timeout, the `INFINITE_TIMEOUT` constant can be specified for these pool parameters. (The `INFINITE_TIMEOUT` constant is defined in the header file `sysroot-*-kos/include/rtl/rtc.h` from the KasperskyOS SDK.)

The functions for creating a pool without the `Ex` postfix use the `KosDispatchPoolDefaultParams` macro to initialize the `KosDispatchPoolParams` structure within these functions. The definition of this macro for initializing the `KosDispatchPoolParams` structure uses symbolic constants in the format `KOS_DISPATCH_POOL_*_DEFAULT` whose values correspond to the values of the default pool parameters.

Functions for creating a pool with the `Ex` postfix receive the `KosDispatchPoolParams` structure via the `params` parameter. The values of pool parameters must not be less than the values of symbolic constants in the format `KOS_DISPATCH_POOL_*_MIN`.

All pool creation functions receive the server IPC handle via the `endpoint` parameter. The threads of the created pool will process IPC requests received over the IPC channels associated with this IPC handle.

Each thread of the pool starts an IPC request processing loop by calling the `NkKosDoDispatchEx()` function from the API transport-kos-dispatch.h. This function requires the `NkKosDispatchInfo` structure whose type is defined in the header file `sysroot-*-kos/include/coresrv/nk/transport-kos-dispatch.h` from the KasperskyOS SDK. Therefore, all pool creation functions receive the pointer to the `NkKosDispatchInfo` structure via the `info` parameter. To initialize this structure, you can use the macros defined in the header file `sysroot-*-kos/include/coresrv/nk/transport-kos-dispatch.h` from the KasperskyOS SDK (for example: `NK_TASK_DISPATCH_INFO_INITIALIZER()`).

All pool creation functions receive the thread stack limit via the `stackSize` parameter, which means that each thread in the pool will have a stack of the defined size. The thread stack limit can be specified by the `ThreadStackSizeDefault` constant that is defined in the header file `sysroot-*-kos/include/coresrv/thread/thread_api.h` from the KasperskyOS SDK. However, you should also keep in mind that the `NkKosDoDispatchEx()` function attempts to create buffers for IPC requests and IPC responses in the stack, and the sizes of these buffers may be a multiple larger than the value of the `ThreadStackSizeDefault` constant. (If the attempt to create a buffer in the stack is unsuccessful, the `NkKosDoDispatchEx()` function attempts to create this buffer in the heap.) If it is more preferable to place buffers for IPC requests and IPC responses in the stack, you must increase the stack limit according to the total size of these buffers.

Deleting a pool

When a pool is deleted, all of its threads are terminated and the resources of the terminated threads are freed. To delete a pool that was created by calling a function with the `Detached` suffix, call the `KosDispatchPoolStop()` function. (When using a function with the `Attached` suffix, you do not need to delete the pool prior to server termination.)

Irrespective of which specific API function was used to create a pool, the pool may be deleted due to an error that occurs when processing an IPC request or when managing the pool.

Information about API functions

dispatch_pool.h functions

Function	Information about the function
`KosDispatchPoolStartAttached()`	Purpose Creates a pool of threads that process IPC requests. The pool is managed by the calling thread. Parameters • [in] `stackSize` – stack limit of each thread in the pool (in bytes). • [in] `endpoint` – server IPC handle. • [in] `info` – pointer to the structure containing the data required by the function to use transport code (including the pointers to the dispatcher and stub, sizes of the constant part and IPC message arena). The type of structure is defined in the header file `sysroot-*-kos/include/coresrv/nk/transport-kos-dispatch.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosDispatchPoolStartAttachedEx()`	Purpose Creates a pool of threads that process IPC requests. The pool is managed by the calling thread. Parameters • [in] `stackSize` – stack limit of each thread in the pool (in bytes). • [in] `endpoint` – server IPC handle. • [in] `info` – pointer to the structure containing the data required by the function to use transport code (including the pointers to the dispatcher and stub, sizes of the constant part and IPC message arena). The type of structure is defined in the header file `sysroot-*-kos/include/coresrv/nk/transport-kos-dispatch.h` from the KasperskyOS SDK. • [in] `params` – structure containing the pool parameters. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KosDispatchPoolStartDetached()`	Purpose Creates a pool of threads that process IPC requests. The pool is managed by a specially created thread. Parameters • [in] `stackSize` – stack limit of each thread in the pool (in bytes). • [in] `endpoint` – server IPC handle. • [in] `info` – pointer to the structure containing the data required by the function to use transport code (including the pointers to the dispatcher and stub, sizes of the constant part and IPC message arena). The type of structure is defined in the header file `sysroot-*-kos/include/coresrv/nk/transport-kos-dispatch.h` from the KasperskyOS SDK. • [out] `outHandle` – pointer to the pool ID. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosDispatchPoolStartDetachedEx()`	Purpose Creates a pool of threads that process IPC requests. The pool is managed by a specially created thread. Parameters • [in] `stackSize` – stack limit of each thread in the pool (in bytes). • [in] `endpoint` – server IPC handle. • [in] `info` – pointer to the structure containing the data required by the function to use transport code (including the pointers to the dispatcher and stub, sizes of the constant part and IPC message arena). The type of structure is defined in the header file `sysroot-*-kos/include/coresrv/nk/transport-kos-dispatch.h` from the KasperskyOS SDK. • [in] `params` – structure containing the pool parameters. • [out] `outHandle` – pointer to the pool ID. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosDispatchPoolStop()`	Purpose Deletes a pool of threads that process IPC requests. The deleted pool was created by calling the `KosDispatchPoolStartDetached()` or `KosDispatchPoolStartDetachedEx()` function. Parameters • [in] `handle` – pool ID. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Generating random numbers (random_api.h)

The API is defined in the header file `sysroot-*-kos/include/kos/random/random_api.h` from the KasperskyOS SDK.

The API lets you generate sequences of random byte values using a random number generator, and includes functions that can be used to ensure high entropy (high level of unpredictability) of the seed value of the random number generator. The *start number of the random number generator* (seed) determines the sequence of the generated random numbers. In other words, if the same seed value is set, the generator creates identical sequences of random numbers. (The entropy of these numbers is fully determined by the entropy of the seed value, which means that these numbers are not entirely random, but pseudorandom.)

The random number generator of one process does not depend on the random number generators of other processes.

Information about API functions is provided in the table below.

Using the API

To generate a sequence of random byte values, call the `KosRandomGenerate()` or `KosRandomGenerateEx()` function.

Example use of the `KosRandomGenerate()` function:

```
size_t random_number;
if (KosRandomGenerate(sizeof random_number, &random_number) == rcOk) {
...
}
```

The `KosRandomGenerateEx()` function gets the quality level of the generated random values through the output parameter `quality`. The quality level can be high or low. High-quality random values are random values that are generated while fulfilling all of the following conditions:

1. When the seed value is changed, at least one entropy source is registered and data is successfully received from all registered entropy sources.

   To register the entropy source, call the `KosRandomRegisterSrc()` function. This function uses the `callback` parameter to receive the pointer to a callback function of the following type:

   ```
   typedef Retcode (*KosRandomSeedMethod)(void      *context,
                                          rtl_size_t size,
                                          void      *output);
   ```

   Using the `context` parameters, this callback function receives data with the specified `size` of bytes from the entropy source and writes this data to the `output` buffer. If the function returns `rcOk`, it is assumed that data was successfully received from the entropy source. An entropy source can be a digitalized signal of a sensor or a hardware-based random number generator, for example.

   To deregister an entropy source, call the `KosRandomUnregisterSrc()` function.

2. The random number generator is initialized if the quality level was low before changing the seed value while fulfilling condition 1.

   If the quality level is low, it cannot be changed to high without initializing the random number generator.

   To initialize a random number generator, call the `KosRandomInitSeed()` function. The entropy of data passed through the `seed` parameter must be guaranteed by the calling process.

3. The counter of random byte values that were generated after changing the seed value while fulfilling condition 1 does not exceed the system-defined limit.

4. The counter of time that has elapsed since changing the seed value while fulfilling condition 1 does not exceed the system-defined limit.

If at least one of these conditions is not fulfilled, the generated random values are deemed low-quality values.

When a process is started, the seed value is automatically defined by the system. Then the seed value is modified when doing the following:

- Generating a sequence of random values.

  Each successful call of the `KosRandomGenerateEx()` function with the `size` parameter greater than zero and each successful call of the `KosRandomGenerate()` function results in a change of the seed value of the random number generator, but not each seed change results in the receipt of data from registered entropy sources. The registered entropy sources are used only when condition 3 or 4 is not fulfilled. If at least one entropy source is registered, the time counter for the change in the seed value is reset. If data from at least one entropy source is successfully received, the counter of generated random byte values is also reset.

  The quality level may change from high to low.

- Initializing a random number generator.

  Each successful call of the `KosRandomInitSeed()` function changes the seed value by using the data that is passed through the `seed` parameter and received from registered entropy sources. If at least one registered entropy source is available, the counters for generated byte values and the time of a change in the seed value are reset. Otherwise, only the counter of generated random byte values is reset.

  The quality level may change from high to low, and vice versa.

- Registering an entropy source.

  Each successful call of the `KosRandomRegisterSrc()` function changes the seed value by using the data only from the registered entropy source. The counter of the generated random byte values is also reset.

  The quality level may not change.

A possible scenario for generating high-quality random values includes the following steps:

1. Register at least one source of entropy by calling the `KosRandomRegisterSrc()` function.

2. Generate a sequence of random values by calling the `KosRandomGenerateEx()` function.

3. Check the quality level of the random values.

   If the quality level is low, initialize the random number generator by calling the `KosRandomInitSeed()` function, and proceed to step 2.

   If the quality level is high, proceed to step 4.

4. Use random values.

To get the quality level without generating random values, call the `KosRandomGenerateEx()` function with the values `0` and `RTL_NULL` in the `size` and `output` parameters, respectively.

Information about API functions

random_api.h functions

Function	Information about the function
`KosRandomInitSeed()`	Purpose Initializes the random number generator. Parameters • [in] `seed` – pointer to the byte array that is used to change the seed value. The array must have a size of 32 bytes. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosRandomGenerate()`	Purpose Generates a sequence of random byte values. Parameters • [in] `size` – size (in bytes) of the buffer used to store the sequence. • [out] `output` – pointer to the buffer used to store the sequence. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosRandomGenerateEx()`	Purpose Generates a sequence of random byte values. Parameters • [in] `size` – size (in bytes) of the buffer used to store the sequence. • [out] `output` – pointer to the buffer used to store the sequence. • [out] `quality` – pointer to the boolean value that is true if the generated random values have high quality, and false if the generated random values have low quality. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosRandomRegisterSrc()`	Purpose Registers an entropy source. Parameters • [in] `callback` – pointer to the function that receives data from the entropy source to change the seed value. • [in,optional] `context` – pointer to the parameters passed to the function defined through the `callback` parameter, or `RTL_NULL` if there are no parameters. • [in] `size` – size (in bytes) of the data that should be received from the entropy source when calling the function defined through the `callback` parameter. The size must be at least 32 bytes. • [out] `handle` – address of the pointer used to deregister the entropy source. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KosRandomUnregisterSrc()`	Purpose Deregisters the source of entropy. Parameters • [in] `handle` – pointer that is received when registering the source of entropy. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Getting and changing time values (time_api.h)

The API is defined in the header file `sysroot-*-kos/include/coresrv/time/time_api.h` from the KasperskyOS SDK.

Main capabilities of the API:

- Get and modify the system time
- Get the monotonic time that has elapsed since the moment the KasperskyOS kernel was started
- Get the resolution of the sources of system and monotonic time
- Receive RTC (Real Time Clock) characteristics

Information about API functions is provided in the table below.

time_api.h functions

Function	Information about the function
`KnGetSystemTimeRes()`	Purpose Gets the resolution of the system time source. Parameters • [out] `res` – pointer to the structure containing the resolution of the system time source (in nanoseconds) in the `nsec` field. The type of structure is defined in the header file `sysroot-*-kos/include/rtl/rtc.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KnSetSystemTime()`	Purpose Sets the system time. Parameters • [in] `time` – pointer to the structure containing the `sec` field, which indicates the number of seconds that have elapsed since January 1, 1970, and the `nsec` field, which indicates the number of nanoseconds that have elapsed since the time defined in the `sec` field. The type of structure is defined in the header file `sysroot-*-kos/include/rtl/rtc.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information If the hardware platform contains an RTC, and the kernel has the required RTC driver, the kernel receives the system time from the RTC at startup. Otherwise, the countdown starts on January 1, 1970. In the second case, the actual system time to be set by the `KnSetSystemTime()` function can be obtained from the RTC driver running in user mode or from the network, for example.
`KnGetSystemTime()`	Purpose Gets the system time. Parameters • [out] `time` – pointer to the structure containing the `sec` field, which indicates the number of seconds that have elapsed since January 1, 1970, and the `nsec` field, which indicates the number of nanoseconds that have elapsed since the time defined in the `sec` field. The type of structure is defined in the header file `sysroot-*-kos/include/rtl/rtc.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnGetUpTimeRes()`	Purpose Gets the resolution of the source of monotonic time that has elapsed since the kernel startup. Parameters • [out] `res` – pointer to a structure whose `nsec` field contains the resolution of the monotonic time source, measured in nanoseconds since the kernel startup. The type of structure is defined in the header file `sysroot-*-kos/include/rtl/rtc.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnGetUpTime()`	Purpose Gets the monotonic time that has elapsed since the kernel startup. Parameters • [out] `time` – pointer to the structure containing the `sec` field, which indicates the number of seconds that have elapsed since the kernel startup, and the `nsec` field, which indicates the number of nanoseconds that have elapsed since the time defined in the `sec` field. The type of structure is defined in the header file `sysroot-*-kos/include/rtl/rtc.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KnGetRtcTime()`	Purpose Gets the system time. Parameters • [out] `rt` – pointer to the structure containing the following time data: year, month, day, hours, minutes, seconds, and milliseconds. The type of structure is defined in the header file `sysroot-*-kos/include/rtl/rtc.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnGetMSecSinceStart()`	Purpose Gets the monotonic time that has elapsed since the kernel startup. Parameters N/A Returned values Monotonic time (in milliseconds) that has elapsed since the kernel startup.
`KnAdjSystemTime()`	Purpose Starts gradual adjustment of the system time. Parameters • [in,optional] `adj` – pointer to the structure containing the amount of time by which the system time must be adjusted (*sec*10^9+nsec* nanoseconds), or `RTL_NULL` if you do not need to start an adjustment but instead only need information about a previously run adjustment (through the `prev` parameter). The type of structure is defined in the header file `sysroot-*-kos/include/rtl/rtc.h` from the KasperskyOS SDK. • [in] `slew` – rate of system time adjustment (microseconds per second). • [out] `prev` – pointer to the structure containing the amount of time correction that remained (or remains if `RTL_NULL` was indicated in the `adj` parameter) for the already running gradual adjustment to fully complete (*sec*10^9+nsec* nanoseconds). The type of structure is defined in the header file `sysroot-*-kos/include/rtl/rtc.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information If a new adjustment is started before a previously running adjustment is finished, the previously running adjustment is interrupted.
`KnGetRtcInfo()`	Purpose Gets RTC characteristics. Parameters • [out] `info` – pointer to a structure containing the RTC characteristics: resolution, and the lower and upper bounds of the countdown time (relative to January 1, 1970). Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. The function returns `rcUnimplemented` if the RTC is not available on the hardware platform, or if the required RTC driver is missing from the kernel.

Using notifications (notice_api.h)

The API is defined in the `sysroot-*-kos/include/coresrv/handle/notice_api.h` header file from the KasperskyOS SDK.

The API can track events associated with (both system and user) resources, and inform other processes and threads about events involving user resources.

The API functions are thread-safe.

Information about API functions is provided in the table below.

Event mask

The notification mechanism uses an event mask. An *event mask* is a value whose bits are interpreted as events that should be tracked or that have already occurred. An event mask has a size of 32 bits and consists of a general part and a specialized part. The common part describes events that are not specific to any resources. The specialized part describes events that are specific to certain resources. Specialized part flags for system resources and common part flags are defined in the `sysroot-*-kos/include/handle/event_descr.h` header file from KasperskyOS SDK. (For example, the common part flag `EVENT_OBJECT_DESTROYED` signifies resource termination, and the specialized part flag `EVENT_TASK_COMPLETED` signifies process termination.) Specialized part flags for a user resource are defined by the resource provider with the help of the `OBJECT_EVENT_SPEC()` and `OBJECT_EVENT_USER()` macros, which are defined in the `sysroot-*-kos/include/handle/event_descr.h` header file from the KasperskyOS SDK. The resource provider must export the public header files describing the flags of the specialized part.

Creating a notification receiver

To create a *notification receiver* (KasperskyOS kernel object that accumulates notifications), call the `KnNoticeCreate()` function.

The notification receiver ID is the pointer to the `KosObject` containing the handle of the notification receiver (for details about `KosObject`s, see Using KosObjects (objects.h)). Destruction of this object will cause the destruction of the notification receiver. Each thread that needs to use a notification receiver created by another thread must increment the `KosObject` reference counter by calling the `KosRefObject()` function to ensure that the notification receiver will exist for the required amount of time. If the notification receiver is no longer needed for this thread, you must decrement the counter of references to the `KosObject` by calling the `KosPutObject()` function to make sure that this object is destroyed if there are no other references.

Configuring a notification receiver

To configure a notification receiver to receive notifications about events that are related to relevant resources, you must add "resource–event mask" entries to this receiver. "Resource—event mask" entries added to the notification receiver can be fully or partially removed to prevent the receiver from getting notifications that match these entries. You can also change the ID of an entry and/or event mask in the "resource–event mask" entries that have been added to the notification receiver. "Resource–event mask" entries can be added, removed, or modified throughout the entire lifetime of the notification receiver.

To add a "resource—event mask" entry to the notification receiver, you need to call the `KnNoticeSubscribeToObject()` or `KnNoticeSubscribeToObjectEx()` function. (The `OCAP_HANDLE_GET_EVENT` flag should be set in the handle permissions mask of the resource stated in the `object` parameter.) Several "resource—event mask" entries can be added for one resource, and the entry identifiers do not need to be unique. Tracked events for each "resource—event mask" entry should be defined with an event mask that may match one or several events. In contrast to the `KnNoticeSubscribeToObject()` function, the `KnNoticeSubscribeToObjectEx()` function lets you add a "resource–event mask" entry that will be associated with the resource handle specified in the `object` parameter and will be automatically removed from the notification receiver when this handle is closed. (The same resource may be identified by multiple handles.) The notification receiver can contain no more than one such entry for each resource handle. You cannot specify the `EVENT_OBJECT_DESTROYED` flag in the event mask of an entry associated with a resource handle.

To change the ID of an entry and/or event mask in "resource–event mask" entries, call the `KnNoticeEditSubscriptionForEvent()` or `KnNoticeEditSubscriptionForObject()` function. The `KnNoticeEditSubscriptionForEvent()` function modifies either all entries with the specified ID or only those entries that correspond to the specified resource. The `KnNoticeEditSubscriptionForObject()` function modifies either all entries that correspond to the specified resource or a single entry associated with the specified resource handle.

To remove all "resource–event mask" entries from the receiver, call the `KnNoticeDropAndWakeEx()` function with the `NoticeRelease` flag in the `flags` parameter, or call the `KnNoticeDropAndWake()` function.

To remove from the receiver "resource—event mask" entries that match the same resource, you need to call the `KnNoticeUnsubscribeFromObject()` or `KnNoticeUnsubscribeFromObjectEx()` function. The `KnNoticeUnsubscribeFromObjectEx()` function is an extended version of the `KnNoticeUnsubscribeFromObject()` function and it additionally lets you delete a single entry associated with the specified resource handle.

To remove from the receiver a "resource—event mask" entry with a specific identifier, you need to call the `KnNoticeUnsubscribeFromEvent()` or `KnNoticeUnsubscribeFromEventEx()` function. The `KnNoticeUnsubscribeFromEventEx()` function is an extended version of the `KnNoticeUnsubscribeFromEvent()` function and it additionally lets you delete only those entries with the specified ID that correspond to the specified resource.

Already received notifications that correspond to modified or deleted "resource–event mask" entries will be automatically deleted from the receiver.

Extracting notifications from the receiver

To extract notifications from the receiver, you must use the `KnNoticeGetEvent()` or `KnNoticeGetEventEx()` function. When calling these functions, you must specify the timeout for notifications to appear in the receiver (the value of 0 is allowed). Threads that are locked while waiting for notifications to appear in the receiver will resume when notifications appear, even if these notifications match "resource—event mask" entries that were added to the receiver during the waiting period.

The `KnNoticeGetEvent()` and `KnNoticeGetEventEx()` functions differ in the fact that they pass different structures via the `events` output parameter. The structures passed by the `KnNoticeGetEventEx()` function contain the additional `eventCounter` field containing an event counter. This counter shows how many events matching the event mask in a notification occurred while this notification resided in the receiver. In other words, the occurrence of an event does not increase the number of notifications in a receiver if this receiver already contains notifications that correspond to the "resource–event mask" entries matching this event. In this case, the event counters are incremented in the corresponding notifications and the event masks may be expanded. A notification event mask can be expanded if this notification corresponds to a "resource–event mask" entry matching multiple events. For example, first a matching event occurred and the receiver received a notification with one event in the event mask and the value of 1 in the event counter, and then another matching event occurred and the event counter was incremented in this notification and the event mask was also expanded.

Threads that are locked while waiting for notifications to appear in the receiver will resume their execution if the `KnNoticeDropAndWake()` or `KnNoticeDropAndWakeEx()` function has been called for this receiver. These functions not only resume execution of threads waiting for the appearance of notifications in the receiver but also deny extraction of notifications from the receiver. If extraction of notifications from the receiver is denied, calling the `KnNoticeGetEvent()` or `KnNoticeGetEventEx()` function does not cause the calling threads to lock, ending instead with the `rcResourceNotFound` error. To reallow extraction of notifications from the receiver, call the `KnNoticeResumeDelivering()` function. The extraction is also reallowed automatically when you add at least one "resource–event mask" entry to the receiver in the following cases:

- Extraction was denied by calling the `KnNoticeDropAndWake()` function.
- Extraction was denied by calling the `KnNoticeDropAndWakeEx()` function with the `NoticeAutoResume` flag in the `flags` parameter.

A call of the `KnNoticeDropAndWakeEx()` function with the `NoticeDefault`, `NoticeAutoResume`, or `NoticeRelease` flag in the `flags` parameter denies extraction of notifications from the receiver. The `NoticeRelease` flag deletes all "resource–event mask" entries and the receiver stops receiving notifications, while the `NoticeDefault` and `NoticeAutoResume` flags do not delete "resource–event mask" entries and the receiver continues to receive notifications. A receiver from which extraction of notifications is banned can be configured – you can add, remove, or modify its "resource–event mask" entries.

If all "resource—event mask" entries are removed from the notification receiver with the `KnNoticeUnsubscribeFromObject()` and/or `KnNoticeUnsubscribeFromEvent()` functions, threads waiting for notifications to appear in the receiver will not resume until the time-out elapses, and notifications extraction will not be denied.

The notifications related to the same resource are arranged as follows: in the receiver, notifications that contain the `EVENT_OBJECT_DESTROYED` flag in their event mask are positioned after notifications that do not contain this flag in their event mask. In other words, notifications with the `EVENT_OBJECT_DESTROYED` flag will be the last notifications in the sequence of notifications extracted from the receiver.

Terminating the use of a notification receiver

To terminate the use of a notification receiver, call the `KnNoticeRelease()` or `KnNoticeDropAndWakeEx()` function with the `NoticeFinalize` and `NoticeRelease` flags in the `flags` parameter. After these calls, the threads that are locked while waiting for notifications to appear in the receiver will resume their execution, and subsequent calls of the `KnNoticeGetEvent()` or `KnNoticeGetEventEx()` function will not lock the calling threads, ending instead with the `rcInvalidOperation` error. You must also keep in mind that the `KnNoticeRelease()` function decrements the counter of references to the `KosObject` containing the handle of the notification receiver. If there are no other links to this object, the notification receiver will be destroyed and subsequent queries to this receiver must be avoided.

When terminating the use of a notification receiver using the `KnNoticeDropAndWakeEx()` function, the counter of references to the `KosObject` containing the notification receiver handle must be decremented by calling the `KosPutObject()` function from the objects.h API.

Signaling about events related to a user resource

To notify other processes and/or threads about events associated with the user resource, you need to call the `KnNoticeSetObjectEvent()` function. Calling the function results in notifications appearing in receivers configured to get events defined with the `evMask` parameter and associated with the user resource defined with the `object` parameter. You cannot set flags of the general part of an event mask in the `evMask` parameter, because only the kernel can signal about events that match the general part of an event mask. If the process calling the `KnNoticeSetObjectEvent()` function created the user resource handle stated in the `object` parameter, you can set flags defined by the `OBJECT_EVENT_SPEC()` and `OBJECT_EVENT_USER()` macros in the `evMask` parameter. If the process calling the `KnNoticeSetObjectEvent()` function received the user resource handle stated in the `object` parameter from another process, you can set only those flags defined by the `OBJECT_EVENT_USER()` macro in the `evMask` parameter, while the permissions mask of the resulting handle must have a `OCAP_HANDLE_SET_EVENT` flag set.

Atomicity of operations

Operations for configuring a notification receiver (adding, deleting, and modifying "resource–event mask" entries) and extraction of notifications from the receiver are performed atomically. The receipt of notifications by a receiver is also performed atomically.

Information about API functions

notice_api.h functions

Function	Information about the function
`KnNoticeCreate()`	Purpose Creates a notification receiver. Parameters • [out] `notice` – pointer to the identifier of the notification receiver. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnNoticeSubscribeToObject()`	Purpose Adds a "resource–event mask" entry to the notification receiver so that it can receive notifications about events associated with the defined resource and match the defined event mask. Parameters • [in] `notice` – identifier of the notification receiver. • [in] `object` – resource handle. • [in] `evMask` – event mask. • [in] `evId` – ID of the "resource–event mask" entry. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnNoticeSubscribeToObjectEx()`	Purpose Adds a "resource–event mask" entry to the notification receiver so that it can receive notifications about events associated with the defined resource and match the defined event mask. Parameters • [in] `notice` – identifier of the notification receiver. • [in] `object` – resource handle. • [in] `evMask` – event mask. • [in] `evId` – ID of the "resource–event mask" entry. • [in] `flags` – flags defining the parameters for the "resource–event mask" entry. The parameter type and flags are defined in the header file `sysroot-*-kos/include/handle/event_descr.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information In the `flags` parameter, you can specify the following flags: • `NoticeDefault` – there are no parameters for the "resource–event mask" entry. • `NoticeAttachedToDescriptor` – associate the "resource–event mask" entry with the resource handle specified in the `object` parameter.

Function	Information about the function
`KnNoticeGetEvent()`	Purpose Extracts notifications from the receiver. Parameters • [in] `notice` – identifier of the notification receiver. • [in] `msec` – timeout (in milliseconds) before notifications appearing in the receiver, or `INFINITE_TIMEOUT` to set an unlimited timeout. The constant `INFINITE_TIMEOUT` is defined in the header file `sysroot-*-kos/include/rtl/rtc.h` from the KasperskyOS SDK. • [in] `countMax` – maximum number of notifications extracted with one function call. • [out] `events` – pointer to a set of notifications that represent structures containing the identifier of a "resource—event mask" entry and the mask of the events associated with the resource. • [out] `count` – number of notifications extracted. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If the timeout for notifications to appear in the receiver has elapsed, returns `rcTimeout`. If extraction of notifications from the receiver has been denied by calling the `KnNoticeDropAndWake()` or `KnNoticeDropAndWakeEx()` function, returns `rcResourceNotFound`. If the notification receiver is disabled as a result of a `KnNoticeRelease()` function call, or `KnNoticeDropAndWakeEx()` returns `rcInvalidOperation`. Additional information Non-blocking call if the `msec` parameter is set to `0`.

Function	Information about the function
`KnNoticeGetEventEx()`	Purpose Extracts notifications from the receiver. Parameters • [in] `notice` – identifier of the notification receiver. • [in] `msec` – timeout (in milliseconds) before notifications appearing in the receiver, or `INFINITE_TIMEOUT` to set an unlimited timeout. The constant `INFINITE_TIMEOUT` is defined in the header file `sysroot-*-kos/include/rtl/rtc.h` from the KasperskyOS SDK. • [in] `countMax` – maximum number of notifications extracted with one function call. • [out] `events` – pointer to a set of notifications that represent structures containing the identifier of a "resource—event mask" entry and the counter of events associated with the resource. • [out] `count` – number of notifications extracted. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If the timeout for notifications to appear in the receiver has elapsed, returns `rcTimeout`. If extraction of notifications from the receiver has been denied by calling the `KnNoticeDropAndWake()` or `KnNoticeDropAndWakeEx()` function, returns `rcResourceNotFound`. If the notification receiver is disabled as a result of a `KnNoticeRelease()` function call, or `KnNoticeDropAndWakeEx()` returns `rcInvalidOperation`. Additional information Non-blocking call if the `msec` parameter is set to `0`.
`KnNoticeUnsubscribeFromObject()`	Purpose Removes from the notification receiver "resource—event mask" all entries that match the specified resource to prevent the receiver from getting notifications about events that match these entries. Parameters • [in] `notice` – identifier of the notification receiver. • [in] `object` – resource handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KnNoticeUnsubscribeFromObjectEx()`	Purpose Removes from the notification receiver "resource—event mask" entries that match the specified resource to prevent the receiver from getting notifications about events that match these entries. Parameters • [in] `notice` – identifier of the notification receiver. • [in] `object` – resource handle. • [in] `flags` – flags defining which "resource–event mask" entries should be deleted. The parameter type and flags are defined in the header file `sysroot-*-kos/include/handle/event_descr.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information In the `flags` parameter, you can specify the following flags: • `NoticeDefault` – delete all "resource–event mask" entries corresponding to the specified resource. • `NoticeAttachedToDescriptor` – delete the "resource–event mask" entry associated with the resource handle specified in the `object` parameter.
`KnNoticeUnsubscribeFromEvent()`	Purpose Removes from the notification receiver "resource—event mask" all entries with the specified identifier to prevent the receiver from getting notifications about events that match these entries. Parameters • [in] `notice` – identifier of the notification receiver. • [in] `evId` – ID of the "resource–event mask" entries. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnNoticeUnsubscribeFromEventEx()`	Purpose Removes from the notification receiver "resource—event mask" entries with the specified identifier to prevent the receiver from getting notifications about events that match these entries. Parameters • [in] `notice` – identifier of the notification receiver. • [in] `evId` – ID of the "resource–event mask" entries. • [in,optional] `object` – resource handle if you need to delete only those "resource–event mask" entries with the specified ID that correspond to the specified resource, or `INVALID_HANDLE` if you need to delete all "resource–event mask" entries with the specified ID. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KnNoticeEditSubscriptionForEvent()`	Purpose Modifies "resource–event mask" entries with the specified ID in the notification receiver. Parameters • [in] `notice` – identifier of the notification receiver. • [in] `evId` – ID of the "resource–event mask" entries that need to be modified. • [in,optional] `object` – resource handle if you need to modify only those "resource–event mask" entries with the specified ID that correspond to the specified resource, or `INVALID_HANDLE` if you need to modify all "resource–event mask" entries with the specified ID. • [in] `newId` – new ID of the "resource–event mask" entries. • [in] `newMask` – new event mask. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnNoticeEditSubscriptionForObject()`	Purpose Modifies the "resource–event mask" entries in the notification receiver that correspond to the specified resource. Parameters • [in] `notice` – identifier of the notification receiver. • [in] `object` – resource handle. • [in] `flags` – flags defining which "resource–event mask" entries should be modified. The parameter type and flags are defined in the header file `sysroot-*-kos/include/handle/event_descr.h` from the KasperskyOS SDK. • [in] `newId` – new ID of the "resource–event mask" entries. • [in] `newMask` – new event mask. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information In the `flags` parameter, you can specify the following flags: • `NoticeDefault` – modify all "resource–event mask" entries corresponding to the specified resource. • `NoticeAttachedToDescriptor` – modify the "resource–event mask" entry associated with the resource handle specified in the `object` parameter.
`KnNoticeDropAndWake()`	Purpose Removes all "resource–event mask" entries from the notification receiver, resumes all threads that are waiting for notifications to appear in this receiver, and denies extraction of notifications from this receiver. Parameters • [in] `notice` – identifier of the notification receiver. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KnNoticeRelease()`	Purpose Disables the notification receiver, resumes all threads waiting for notifications to appear in this receiver, and decrements the counter of references to the `KosObject` containing the handle of this notification receiver. Parameters • [in] `notice` – identifier of the notification receiver. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnNoticeSetObjectEvent()`	Purpose Signals the occurrence of events that are related to the defined user resource and match the defined event mask. Parameters • [in] `object` – user resource handle. • [in] `evMask` – mask of events to be signaled. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnNoticeDropAndWakeEx()`	Purpose Resumes execution of all threads that are waiting for notifications to appear in the receiver and performs additional actions depending on the `flags` parameter. Parameters • [in] `notice` – identifier of the notification receiver. • [in] `flags` – flags defining additional actions. The parameter type and flags are defined in the header file `sysroot-*-kos/include/handle/event_descr.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information In the `flags` parameter, you can specify the following flags: • `NoticeDefault` – denies extraction of notifications from the receiver; extraction is reallowed by calling the `KnNoticeResumeDelivering()` function. • `NoticeAutoResume` – denies extraction of notifications from the receiver; extraction is reallowed automatically when you add a "resource–event mask" entry, it can also be reallowed by calling the `KnNoticeResumeDelivering()` function. • `NoticeRelease` – denies extraction of notifications from the receiver; extraction is reallowed by calling the `KnNoticeResumeDelivering()` function, all "resource–event mask" entries and already received notifications are removed from the receiver. • `NoticeFinalize` – disables the notification receiver. It must be specified together with the `NoticeRelease` flag.

Function	Information about the function
`KnNoticeResumeDelivering()`	Purpose Reallows extraction of notifications from the receiver that was denied by calling the `KnNoticeDropAndWake()` or `KnNoticeDropAndWakeEx()` function. Parameters • [in] `notice` – identifier of the notification receiver. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Using synchronization primitives (event.h, mutex.h, rwlock.h, semaphore.h, condvar.h, sync_api.h, sync_types.h, sync.h)

The APIs are intended for synchronizing threads that belong to the same process.

Using events (event.h)

The API is defined in the header file `sysroot-*-kos/include/kos/event.h` from the KasperskyOS SDK. The KasperskyOS SDK also provides the header file `sysroot-*-kos/include/kos/sync_api.h`, which lets you transitively include the header file `event.h` and the header files for using other synchronization primitives into the application source code.

The API facilitates using events to synchronize threads. An *event* is a synchronization primitive that is used to notify one or more threads about the fulfillment of a condition required by these threads. The notified thread waits for the event to switch from a non-signaling state to a signaling state, and the notifying thread changes the state of this event.

Information about API functions is provided in the table below.

Using the API

The standard scenario for API usage includes the following steps:

1. An event is initialized via the `KosEventInit()` function call.
2. The event is used by threads:
   - The notified threads wait for the event to switch from non-signaling state to signaling state via the `KosEventWait()` or `KosEventWaitTimeout()` function call.
   - The notifying threads change the state of the event via the `KosEventSet()` and `KosEventReset()` function calls.

Information about API functions

event.h functions

Function	Information about the function
`KosEventInit()`	Purpose Initializes an event. The event is in a non-signaling state after it is initialized. Parameters • [out] `event` – pointer to the event. The event type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values N/A
`KosEventSet()`	Purpose Sets the event state to signaling. Parameters • [out] `event` – pointer to the event. The event type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values N/A
`KosEventReset()`	Purpose Sets the event state to non-signaling. Parameters • [out] `event` – pointer to the event. The event type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values N/A
`KosEventWait()`	Purpose Waits for the event to change its state from non-signaling to signaling. Parameters • [in,out] `event` – pointer to the event. The event type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. • [in] `reset` – boolean value that defines whether the event state should be changed set to non-signaling after the time-out has elapsed (`rtl_true` – yes, `rtl_false` – no). The parameter type is defined in the `sysroot-*-kos/include/rtl/stdbool.h` header file from the KasperskyOS SDK. Returned values N/A

Function	Information about the function
`KosEventWaitTimeout()`	Purpose Waits on the event to change its state from non-signaling to signaling for a period that does not exceed the specified time. Parameters • [in,out] `event` – pointer to the event. The event type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. • [in] `reset` – Boolean value that specifies whether the event state should be set to non-signaling if the wait was terminated due to a transition to the signaling state(`rtl_true` means yes, `rtl_false` means no). The parameter type is defined in the `sysroot-*-kos/include/rtl/stdbool.h` header file from the KasperskyOS SDK. • [in] `mdelay` – timeout (in milliseconds) or `INFINITE_TIMEOUT` to set an unlimited timeout. The constant `INFINITE_TIMEOUT` is defined in the header file `sysroot-*-kos/include/rtl/rtc.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Returns `rcTimeout` if the timeout has elapsed. Additional information Non-blocking call if the `mdelay` parameter is set to `0`.

Using mutexes (mutex.h)

The API is defined in the header file `sysroot-*-kos/include/kos/mutex.h` from the KasperskyOS SDK. The KasperskyOS SDK also provides the header file `sysroot-*-kos/include/kos/sync_api.h`, which lets you transitively include the header file `mutex.h` and the header files for using other synchronization primitives into the application source code.

The API facilitates using mutexes to synchronize threads.A *mutex* is a synchronization primitive that ensures mutually exclusive execution of *critical sections* (areas of code where resources shared between threads are queried). One thread captures the mutex and executes a critical section. Meanwhile, other threads wait for the mutex to be freed and attempt to capture this mutex to execute other critical sections. A mutex can be freed only by the specific thread that captured it. A *recursive mutex* can be captured by a single thread multiple times.

You can use mutexes without priority inheritance support and optionally (depending on the configuration of the KasperskyOS kernel) with priority inheritance support.

Mutexes with priority inheritance support have the following features:

- Threads that are locked while waiting for the release of the mutex form a queue that consists of two parts. The initial part of the queue consists of real-time threads (scheduler classes FIFO and RR) arranged in the order of their priority. The final part of the queue consists of standard threads arranged according to the FIFO principle. The higher the priority of a real-time thread, the closer this thread is to the start of the queue. In this case, real-time threads with the same priority are ordered according to the FIFO principle. (For scheduler classes and thread priorities, see Managing threads (low-level API thread_api.h)).

- Capturing a mutex can change the scheduler class and priority of a thread. The change is in effect until the mutex is released, after which the scheduler class and priority are restored to their previous values. The purpose of these changes is to prevent high-priority threads from being locked for prolonged periods while waiting for the release of mutexes captured by low-priority threads.

    - If the first in the queue is a real-time thread with a higher priority than that of the real-time thread that captured the mutex, the scheduler class of the thread that captured the mutex is changed to FIFO (if it was RR) while the priority of the thread is raised to the priority of the first thread in the queue.

    - If the first thread in the queue is a real-time thread, the scheduler class of the standard thread that has captured the mutex is changed to FIFO while the priority of the thread becomes equal to the priority of the first thread in the queue.

    - If the standard thread is the first in the queue, the scheduler class and priority of the thread that captured the mutex are not changed.

    - If a real-time thread whose scheduler class and/or priority has changed as a result of the capture of mutex 1 is locked while awaiting the release of mutex 2, and mutex 2 is captured by a real-time thread with a lower priority than the temporary priority of the thread that captured mutex 1, the scheduler class of the thread that captured mutex 2 is changed to FIFO (if it was RR) while the priority of the thread is raised to the temporary priority of the thread that captured mutex 1.

    - If a real-time thread whose scheduler class and/or priority has changed as a result of the capture of mutex 1 is locked while awaiting the release of mutex 2, and mutex 2 is captured by a standard thread, the scheduler class of the thread that captured mutex 2 is changed to FIFO while the priority of the thread becomes equal to the temporary priority of the thread that captured mutex 1.

- Mutexes cannot be recursive.

Mutexes without priority inheritance support have the following special features:

- Threads that are locked while waiting for the release of the mutex form a FIFO-ordered queue, regardless of scheduler classes and priorities.

- The scheduler class and priority of a thread cannot be changed as a result of mutex capture.

- Mutexes can be recursive.

Information about API functions is provided in the table below.

Using the API

The standard scenario for API usage includes the following steps:

1. Initialize the mutex by calling the `KosMutexInit()`, `KosMutexInitEx()`, or `KosRtMutexInit()` function.

   If you need to use mutexes that support priority inheritance, you must check whether the version of the kernel you are using supports this feature before performing this step. To do so, call the `KosRtMutexIsSupported()` function.

2. The mutex is used by threads:

   a. Capture the mutex by calling a function from the following set: `Kos(Rt)MutexTryLock()`, `Kos(Rt)MutexLock()`, `Kos(Rt)MutexLockTimeout()`.

   b. Release the mutex by calling the `KosMutexUnlock()` or `KosRtMutexUnlock()` function.

   c. Check the state of the mutex (whether it is captured or not) by calling the `KosMutexIsLocked()` or `KosRtMutexIsLocked()` function.

      The state of the mutex may change immediately after the `Kos(Rt)MutexIsLocked()` functions return control. These functions can be used during debugging, for example, to ensure that the mutex is in the expected state.

Information about API functions

mutex.h functions

Function	Information about the function
`KosMutexInit()`	Purpose Initializes a mutex that has no priority inheritance support and is not recursive. Parameters • [out] `mutex` – pointer to the mutex. The mutex type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values N/A
`KosMutexInitEx()`	Purpose Initializes a mutex without priority inheritance support. Parameters • [out] `mutex` – pointer to the mutex. The mutex type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. • [in] `recursive` – value that defines whether the mutex should be recursive (`1` – yes, `0` – no). Returned values N/A

Function	Information about the function
`KosMutexTryLock()`	Purpose Captures a mutex without priority inheritance support. If the mutex is already acquired, returns control rather than waits for the mutex to be released. Parameters • [in,out] `mutex` – pointer to the mutex. The mutex type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If the mutex is already acquired, returns `rcBusy`.
`KosMutexLock()`	Purpose Captures a mutex without priority inheritance support. If the mutex is already acquired, waits indefinitely for it to be released. Parameters • [in,out] `mutex` – pointer to the mutex. The mutex type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values N/A
`KosMutexUnlock()`	Purpose Frees a mutex that has no priority inheritance support. Parameters • [in,out] `mutex` – pointer to the mutex. The mutex type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values N/A
`KosMutexLockTimeout()`	Purpose Captures a mutex without priority inheritance support. If the mutex is already acquired, waits for it to be released for a period that does not exceed the specified time. Parameters • [in,out] `mutex` – pointer to the mutex. The mutex type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. • [in] `mdelay` – timeout (in milliseconds) or `INFINITE_TIMEOUT` to set an unlimited timeout. The constant `INFINITE_TIMEOUT` is defined in the header file `sysroot-*-kos/include/rtl/rtc.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Returns `rcTimeout` if the timeout has elapsed. Additional information Non-blocking call if the `mdelay` parameter is set to `0`.

Function	Information about the function
`KosMutexIsLocked()`	Purpose Checks whether a mutex without priority inheritance support has been captured. Parameters • [in] `mutex` – pointer to the mutex. The mutex type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values: If the mutex has been acquired, the function returns `rtl_true`. Otherwise it returns `rtl_false`. The type of returned value is defined in the header file `sysroot-*-kos/include/rtl/stdbool.h` from the KasperskyOS SDK.
`KosRtMutexIsSupported()`	Purpose Checks whether the version of the kernel in use provides the capability to use mutexes with priority inheritance support. Parameters N/A Returned values If the verification is successful, the function returns `rtl_true`, otherwise it returns `rtl_false`. The type of returned value is defined in the header file `sysroot-*-kos/include/rtl/stdbool.h` from the KasperskyOS SDK.
`KosRtMutexInit()`	Purpose Initializes a mutex with priority inheritance support. Parameters • [out] `mutex` – pointer to the mutex. The mutex type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values N/A
`KosRtMutexTryLock()`	Purpose Captures a mutex with priority inheritance support. If the mutex is already acquired, returns control rather than waits for the mutex to be released. Parameters • [in,out] `mutex` – pointer to the mutex. The mutex type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If the mutex is already acquired, returns `rcBusy`.

Function	Information about the function
`KosRtMutexLock()`	Purpose Captures a mutex with priority inheritance support. If the mutex is already acquired, waits indefinitely for it to be released. Parameters • [in,out] `mutex` – pointer to the mutex. The mutex type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values N/A
`KosRtMutexUnlock()`	Purpose Frees a mutex with priority inheritance support. Parameters • [in,out] `mutex` – pointer to the mutex. The mutex type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values N/A
`KosRtMutexLockTimeout()`	Purpose Captures a mutex with priority inheritance support. If the mutex is already acquired, waits for it to be released for a period that does not exceed the specified time. Parameters • [in,out] `mutex` – pointer to the mutex. The mutex type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. • [in] `mdelay` – timeout (in milliseconds) or `INFINITE_TIMEOUT` to set an unlimited timeout. The constant `INFINITE_TIMEOUT` is defined in the header file `sysroot-*-kos/include/rtl/rtc.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Returns `rcTimeout` if the timeout has elapsed. Additional information Non-blocking call if the `mdelay` parameter is set to `0`.
`KosRtMutexIsLocked()`	Purpose Checks whether a mutex with priority inheritance support has been captured. Parameters • [in] `mutex` – pointer to the mutex. The mutex type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values: If the mutex has been acquired, the function returns `rtl_true`. Otherwise it returns `rtl_false`. The type of returned value is defined in the header file `sysroot-*-kos/include/rtl/stdbool.h` from the KasperskyOS SDK.

Using read-write locks (rwlock.h)

The API is defined in the header file `sysroot-*-kos/include/kos/rwlock.h` from the KasperskyOS SDK. The KasperskyOS SDK also provides the header file `sysroot-*-kos/include/kos/sync_api.h`, which lets you transitively include the header file `rwlock.h` and the header files for using other synchronization primitives into the application source code.

The API facilitates using read-write locks to synchronize threads. A *read-write lock* is a synchronization primitive used to allow access to resources shared between threads: write access for one thread or read access for multiple threads at the same time.

Information about API functions is provided in the table below.

Using the API

The standard scenario for API usage includes the following steps:

1. A read-write lock is initialized by the `KosRWLockInit()` function call.
2. The read-write lock is used by threads:
   a. The read-write lock is captured for write operations (via the `KosRWLockWrite()` or `KosRWLockTryWrite()` function call) or for read operations (via the `KosRWLockRead()` or `KosRWLockTryRead()` function call).
   b. The read-write lock is freed via the `KosRWLockUnlock()` function call.

Information about API functions

rwlock.h functions

Function	Information about the function
`KosRWLockInit()`	Purpose Initializes a read-write lock. Parameters • [out] `rwlock` – pointer to a read-write lock. The read-write lock type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values N/A

Function	Information about the function
`KosRWLockRead()`	Purpose Acquires the read-write lock for reading. If the read-write lock is already acquired for writing, or if there are threads waiting on the lock to be acquired for writing, waits indefinitely for the lock to be released. Parameters • [in,out] `rwlock` – pointer to the read-write lock. The read-write lock type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values N/A
`KosRWLockTryRead()`	Purpose Acquires the read-write lock for reading. If the read-write lock is already acquired for writing, or if there are threads waiting on the lock to be acquired for writing, returns control, rather than waits for the lock to be released. Parameters • [in,out] `rwlock` – pointer to the read-write lock. The read-write lock type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosRWLockWrite()`	Purpose Acquires the read-write lock for writing. If the read-write lock is already acquired for writing or reading, waits indefinitely for the lock to be released. Parameters • [in,out] `rwlock` – pointer to the read-write lock. The read-write lock type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values N/A
`KosRWLockTryWrite()`	Purpose Acquires the read-write lock for writing. If the read-write lock is already acquired for writing or reading, returns control, rather than waits for the lock to be released. Parameters • [in,out] `rwlock` – pointer to the read-write lock. The read-write lock type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KosRWLockUnlock()`	Purpose Releases the read-write lock. Parameters • [in,out] `rwlock` – pointer to the read-write lock. The read-write lock type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values N/A Additional information If the read-write lock is acquired for reading, it remains acquired for reading until released by every reading thread.

Using semaphores (semaphore.h)

The API is defined in the header file `sysroot-*-kos/include/kos/semaphore.h` from the KasperskyOS SDK. The KasperskyOS SDK also provides the header file `sysroot-*-kos/include/kos/sync_api.h`, which lets you transitively include the header file `semaphore.h` and the header files for using other synchronization primitives into the application source code.

The API facilitates using semaphores to synchronize threads. A *semaphore* is a synchronization primitive that is based on a counter whose value can be atomically modified. The value of the counter normally reflects the number of available resources shared between threads. To execute a critical section, the thread waits until the counter value becomes greater than zero. If the counter value is greater than zero, it is decremented by one and the thread executes the critical section. After the critical section is executed, the thread signals the semaphore and the counter value is increased.

Information about API functions is provided in the table below.

Using the API

The standard scenario for API usage includes the following steps:

1. A semaphore is initialized via the `KosSemaphoreInit()` function call.
2. The semaphore is used by threads:
   a. They wait for the semaphore via the `KosSemaphoreWait()`, `KosSemaphoreWaitTimeout()` or `KosSemaphoreTryWait()` function call.
   b. The semaphore is signaled via the `KosSemaphoreSignal()` or `KosSemaphoreSignalN()` function call.
3. Deallocating semaphore resources by calling the `KosSemaphoreDeinit()` function.

Information about API functions

semaphore.h functions

Function	Information about the function
`KosSemaphoreInit()`	Purpose Initializes a semaphore. Parameters • [out] `semaphore` – pointer to the semaphore. The semaphore type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. • [in] `count` – counter value. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If the value of the `count` parameter exceeds the `KOS_SEMAPHORE_VALUE_MAX` constant, returns `rcInvalidArgument`. (The `KOS_SEMAPHORE_VALUE_MAX` constant is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK.)
`KosSemaphoreDeinit()`	Purpose Deallocates semaphore resources. Parameters • [in] `semaphore` – pointer to the semaphore. The semaphore type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If there are threads waiting on the semaphore, returns `rcBusy`.
`KosSemaphoreSignal()`	Purpose Signals the semaphore and increases the counter by one. Parameters • [in, out] `semaphore` – pointer to the semaphore. The semaphore type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosSemaphoreSignalN()`	Purpose Signals the semaphore and increases the counter by the specified number. Parameters • [in, out] `semaphore` – pointer to the semaphore. The semaphore type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. • [in] `n` – natural number by which to increase the counter. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KosSemaphoreWaitTimeout()`	Purpose Waits on the semaphore for a period that does not exceed the specified time. Parameters • [in, out] `semaphore` – pointer to the semaphore. The semaphore type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. • [in] `mdelay` – semaphore timeout (in milliseconds) or `INFINITE_TIMEOUT` to set an unlimited timeout. The constant `INFINITE_TIMEOUT` is defined in the header file `sysroot-*-kos/include/rtl/rtc.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Returns `rcTimeout` if the timeout has elapsed. Additional information Non-blocking call if the `mdelay` parameter is set to `0`.
`KosSemaphoreWait()`	Purpose Waits on the semaphore indefinitely. Parameters • [in, out] `semaphore` – pointer to the semaphore. The semaphore type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosSemaphoreTryWait()`	Purpose Waits on the semaphore. If the semaphore counter is zero, returns control, rather than waits for the semaphore counter to increase. Parameters • [in, out] `semaphore` – pointer to the semaphore. The semaphore type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If the semaphore counter has a zero value, returns `rcBusy`.

Using condition variables (condvar.h)

The API is defined in the header file `sysroot-*-kos/include/kos/condvar.h` from the KasperskyOS SDK. The KasperskyOS SDK also provides the header file `sysroot-*-kos/include/kos/sync_api.h`, which lets you transitively include the header file `condvar.h` and the header files for using other synchronization primitives into the application source code.

The API facilitates using condition variables to synchronize threads. A *condition variable* is a synchronization primitive that is used to notify one or more threads about the fulfillment of a condition required by these threads. A conditional variable is used together with a mutex. The notifying and notified threads capture a mutex to execute critical sections. During execution of a critical section, the notified thread verifies that its required condition was fulfilled (for example, the data has been prepared by the notifying thread). If the condition is fulfilled, the notified thread executes the critical section and frees the mutex. If the condition is not fulfilled, the notified thread is locked at the condition variable and waits for the condition to be fulfilled. When this happens, the mutex is automatically freed. During execution of a critical section, the notifying thread verifies fulfillment of the condition required by the notified thread. If the condition is fulfilled, the notifying thread signals this fulfillment through the condition variable and frees the mutex. The notified thread that was locked and waiting for the fulfillment of its required condition resumes execution of the critical section while automatically capturing the mutex. After the critical section is executed, the notified thread frees the mutex.

Information about API functions is provided in the table below.

Using the API

The standard scenario for API usage includes the following steps:

1. The condition variable and mutex are initialized.

   To initialize a condition variable, you need to call the `KosCondvarInit()` function.

2. The condition variable and mutex are used by threads.

Use of a condition variable and mutex by notified threads includes the following steps:

1. The mutex is captured.
2. Condition fulfillment is verified.
3. Wait for the condition to be fulfilled by calling the `Kos(Rt)CondvarWait()` or `Kos(Rt)CondvarWaitTimeout()` function.

   After the `Kos(Rt)CondvarWait()` or `Kos(Rt)CondvarWaitTimeout()` function is returned, normally you need to re-verify that the condition is fulfilled because another notified thread also received the signal and may have voided this condition again. (For example, another thread could have extracted the data prepared by the notifying thread). If a second check is required, you need to use the following construct to complete steps 2 and 3:

   ```
   while(<condition is not fulfilled>)
   <call Kos(Rt)CondvarWait() or Kos(Rt)CondvarWaitTimeout()>
   ```

4. The mutex is freed.

Use of a condition variable and mutex by notifying threads includes the following steps:

1. The mutex is captured.
2. Condition fulfillment is verified.

3. Fulfillment of the condition is signaled via the `KosCondvarSignal()` or `KosCondvarBroadcast()` function call.

4. The mutex is freed.

Information about API functions

condvar.h functions

Function	Information about the function
`KosCondvarInit()`	Purpose Initializes a condition variable. Parameters • [out] `condvar` – pointer to the condition variable. The conditional variable type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values N/A
`KosCondvarWaitTimeout()`	Purpose Waits for condition fulfillment for a period that does not exceed the specified time. Parameters • [in] `condvar` – pointer to the condition variable. The conditional variable type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. • [in,out] `mutex` – pointer to a mutex with no support for priority inheritance. The mutex type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. • [in] `mdelay` – condition fulfillment timeout (in milliseconds), or `INFINITE_TIMEOUT` to set an unlimited timeout. The constant `INFINITE_TIMEOUT` is defined in the header file `sysroot-*-kos/include/rtl/rtc.h` from the KasperskyOS SDK. Returned values Returns `rcOk` if successful. Returns `rcTimeout` if the timeout has elapsed. Additional information Non-blocking call if the `mdelay` parameter is set to `0`.
`KosCondvarWait()`	Purpose Waits indefinitely for condition fulfillment. Parameters • [in] `condvar` – pointer to the condition variable. The conditional variable type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. • [in,out] `mutex` – pointer to a mutex with no support for priority inheritance. The mutex type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KosRtCondvarWaitTimeout()`	Purpose Waits for condition fulfillment for a period that does not exceed the specified time. Parameters • [in] `condvar` – pointer to the condition variable. The conditional variable type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. • [in,out] `mutex` – pointer to a mutex with support for priority inheritance. The mutex type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. • [in] `mdelay` – condition fulfillment timeout (in milliseconds), or `INFINITE_TIMEOUT` to set an unlimited timeout. The constant `INFINITE_TIMEOUT` is defined in the header file `sysroot-*-kos/include/rtl/rtc.h` from the KasperskyOS SDK. Returned values Returns `rcOk` if successful. Returns `rcTimeout` if the timeout has elapsed. Additional information Non-blocking call if the `mdelay` parameter is set to `0`.
`KosRtCondvarWait()`	Purpose Waits indefinitely for condition fulfillment. Parameters • [in] `condvar` – pointer to the condition variable. The conditional variable type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. • [in,out] `mutex` – pointer to a mutex with support for priority inheritance. The mutex type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosCondvarSignal()`	Purpose Signals condition fulfillment to one of the threads waiting for it. Parameters • [in, out] `condvar` – pointer to the condition variable. The conditional variable type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values N/A
`KosCondvarBroadcast()`	Purpose Signals condition fulfillment to all threads waiting for it. Parameters • [in, out] `condvar` – pointer to the condition variable. The conditional variable type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values N/A

Static initialization of synchronization primitives (sync_types.h)

The API is defined in the header file `sysroot-*-kos/include/kos/sync_types.h` from the KasperskyOS SDK.

The API lets you initialize the following synchronization primitives during compilation:

- mutexes
- events
- semaphores
- condition variables
- read-write locks

Information about API macros is provided in the table below.

Example of static initialization of a mutex:

```
static KosMutex mmap_lock = KosInitializedMutex;
```

sync_types.h macros

Macro	Macro details
`KosInitializedMutexEx()`	Purpose Initializes a mutex without priority inheritance support. Parameters • [in] `_recursive` – value that defines whether the mutex should be recursive (`1` – yes, `0` – no). Macro values Code initializing a mutex.
`KosInitializedMutex`	Purpose Initializes a mutex that has no priority inheritance support and is not recursive. Parameters N/A Macro values Code initializing a mutex.
`KosInitializedRtMutex`	Purpose Initializes a mutex with priority inheritance support. Parameters N/A Macro values Code initializing a mutex.

Macro	Macro details
`KosInitializedRecursiveMutex`	Purpose Initializes a recursive mutex. Parameters N/A Macro values Code initializing a mutex.
`KosInitializedEvent`	Purpose Initializes an event. The event is in a non-signaling state after it is initialized. Parameters N/A Macro values Code initializing an event.
`KosInitializedSemaphore`	Purpose Initializes a semaphore. Parameters N/A Macro values Code initializing a semaphore.
`KosInitializedCondvar`	Purpose Initializes a condition variable. Parameters N/A Macro values Code initializing a condition variable.
`KosInitializedRWLock`	Purpose Initializes a read-write lock. Parameters N/A Macro values Code initializing a read-write lock.

Using futexes (sync.h)

This API is defined in the header file `sysroot-*-kos/include/coresrv/sync/sync.h` from the KasperskyOS SDK.

The API is intended to perform futex-based thread synchronization and is used in the code of the API functions event.h, mutex.h, rwlock.h, semaphore.h, and condvar.h. A *futex* is a low-level synchronization primitive that supports two operations: lock a thread and add it to the futex-linked queue of locked threads, and resume execution of threads from the futex-linked queue of locked threads. A futex is a kernel object linked to an integer variable in the user space. The kernel object provides the capability to store a queue of locked threads linked to the futex. The value of the integer variable in the user space (futex value) is atomically modified by the synchronized threads to signal the changed state of shared resources. For example, the futex value may indicate the specific state of an event (signaling or non-signaling state), indicate whether the mutex has been captured or freed, and indicate the specific value of the semaphore counter.

Two types of futexes are supported:

- Futex linked to a queue of locked threads arranged in the order of their priority.

  Locked threads form a queue consisting of two parts. The initial part of the queue consists of real-time threads (scheduler classes FIFO and RR) arranged in the order of their priority. The final part of the queue consists of standard threads arranged according to the FIFO principle. The higher the priority of a real-time thread, the closer this thread is to the start of the queue. In this case, real-time threads with the same priority are ordered according to the FIFO principle. (For scheduler classes and thread priorities, see Managing threads (low-level API thread_api.h)).

  The futex value must adhere to a special format.

- Futex linked to a queue of locked threads arranged according to the FIFO principle.

  Locked threads form a FIFO-ordered queue, regardless of scheduler classes and priorities.

  The futex value can be an arbitrary non-negative integer.

Information about API functions is provided in the table below.

Using the API

To use a futex linked to a queue of locked FIFO-ordered threads, you only need to create an integer variable to store its value. The `KnFutexWait()` and `KnFutexWake()` functions receive the pointer to this variable via the `ftx` parameter. The kernel object linked to this variable is created or deleted automatically when using these functions.

The `KnFutexWait()` function locks the calling thread if the futex value matches the value of the `val` parameter. (A futex value can be changed by another thread during execution of a function.) The thread is locked for the `mdelay` time period, but execution of this thread may be resumed before the `mdelay` period elapses by calling the `KnFutexWake()` function from another thread. The `KnFutexWake()` function resumes execution of threads from a futex-linked queue of locked threads. The number of threads whose execution is resumed is limited by the value of the `nThreads` parameter. Thread execution is resumed starting with the beginning of the queue.

When using a futex linked to a queue of locked threads ordered by priority, the `KnFutexPiWait()` and `KnFutexPiWake()` functions are used to lock and resume execution of threads, respectively. These functions are intended for use by the `libkos` library. The `libkos` library uses them to implement functions for synchronizing threads based on mutexes that support priority inheritance (`KosRtMutex*()` functions from the mutex.h API).

Information about API functions

sync.h functions

Function	Information about the function
`KnFutexWait()`	Purpose Blocks the calling thread if the futex value is equal to the expected value. Parameters • [in] `ftx` – pointer to the variable containing the futex value. • [in] `val` – expected value of the futex. • [in] `mdelay` – maximum blocking duration (in milliseconds), or `INFINITE_TIMEOUT` to set an unlimited blocking duration. The constant `INFINITE_TIMEOUT` is defined in the header file `sysroot-*-kos/include/rtl/rtc.h` from the KasperskyOS SDK. • [out,optional] `outDelay` – actual blocking duration (in milliseconds), or `RTL_NULL` if this information is not required. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If the thread blocking duration has expired, the function returns `rcTimeout`. If the futex value is not equal to the expected value, the function returns `rcFutexWouldBlock`. Additional information Non-blocking call if the `mdelay` parameter is set to `0`.
`KnFutexWake()`	Purpose Resumes execution of the threads that are the first in the queue of threads that were locked by calls of the `KnFutexWait()` function with the specified futex. Parameters • [in] `ftx` – pointer to the variable containing the futex value. • [in] `nThreads` – maximum number of threads whose execution will be resumed. • [out,optional] `wokenCnt` – actual number of threads whose execution has been resumed, or `RTL_NULL` if this information is not required. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KnFutexPiWait()`	Purpose Locks the calling thread. Parameters • [in] `ftx` – pointer to the variable containing the futex value. The variable type is defined in the header file `sysroot-*-kos/include/rtl/atomic.h` from the KasperskyOS SDK. • [in] `mdelay` – maximum blocking duration (in milliseconds), or `INFINITE_TIMEOUT` to set an unlimited blocking duration. The constant `INFINITE_TIMEOUT` is defined in the header file `sysroot-*-kos/include/rtl/rtc.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information Non-blocking call if the `mdelay` parameter is set to `0`.
`KnFutexPiWake()`	Purpose Resumes execution of the thread that is the first in the queue of threads that were locked by calls of the `KnFutexPiWait()` function with the specified futex. Parameters • [in] `ftx` – pointer to the variable containing the futex value. The variable type is defined in the header file `sysroot-*-kos/include/rtl/atomic.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Managing I/O memory isolation (iommu_api.h)

The API is defined in the `sysroot-*-kos/include/coresrv/iommu/iommu_api.h` header file from the KasperskyOS SDK.

The API is intended for managing the isolation of physical memory regions used by devices on a PCIe bus for DMA. (Isolation is provided by the IOMMU.)

Information about API functions is provided in the table below.

Using the API

A device on the PCIe bus cannot use DMA unless the device is attached to the IOMMU domain. After a device is attached to the IOMMU domain, the device can access all DMA buffers that are associated with this IOMMU domain. A device can be attached to only one IOMMU domain at a time, but multiple devices can be attached to the same IOMMU domain. A DMA buffer can be associated with multiple IOMMU domains at the same time. Each IOMMU domain is associated with only one process, but multiple IOMMU domains can be associated with the same process. The capability to create multiple IOMMU domains associated with the same process allows you to restrict access to DMA buffers for different devices managed by the same process.

The API lets you create IOMMU domains, attach devices on a PCIe bus to IOMMU domains, and detach devices on the PCIe bus from IOMMU domains. A device is normally attached to an IOMMU domain when its driver is initialized. A device is usually detached from an IOMMU domain when errors are encountered during driver initialization or driver finalization.

A DMA buffer is associated with an IOMMU domain when calling the `KnIoDmaBegin()` and `KnIoDmaBeginEx()` functions that are included in the dma.h API. The `KnIoDmaBegin()` function associates a DMA buffer with an automatically created IOMMU domain. Each process may be associated with only one such IOMMU domain. This IOMMU domain is created after the first successful call of the `KnIommuAttachDevice()` function and exists throughout the entire lifetime of the process. The `KnIoDmaBeginEx()` function associates a DMA buffer with the IOMMU domain that was created by calling the `KnIommuCreateDomain()` function. Each process may be associated with multiple IOMMU domains that were created by `KnIommuCreateDomain()` function calls. These IOMMU domains may be removed prior to process termination by closing their handles.

To attach a device to an automatically created IOMMU domain or detach it, you must call the `KnIommuAttachDevice()` or `KnIommuDetachDevice()` function, respectively.

To attach a device to an IOMMU domain that was created by calling the `KnIommuCreateDomain()` function or detach it, you must call the `KnIommuAttachDeviceToDomain()` or `KnIommuDetachDeviceFromDomain()` function, respectively.

Information about API functions

iommu_api.h functions

Function	Information about the function
`KnIommuAttachDevice()`	Purpose Attaches a device on a PCIe bus to the automatically created IOMMU domain associated with the calling process. Parameters • [in] `bdf` – address of the device on the PCIe bus in BDF format. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information If IOMMU is not enabled, `rcOk` is returned.
`KnIommuDetachDevice()`	Purpose Detaches a device on a PCIe bus from the automatically created IOMMU domain associated with the calling process. Parameters • [in] `bdf` – address of the device on the PCIe bus in BDF format. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information If IOMMU is not enabled, `rcOk` is returned.

Function	Information about the function
`KnIommuCreateDomain()`	Purpose Creates an IOMMU domain associated with the calling process. Parameters • [out] `domain` – pointer to the IOMMU domain handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnIommuAttachDeviceToDomain()`	Purpose Attaches a device on a PCIe bus to the defined IOMMU domain associated with the calling process. Parameters • [in] `bdf` – address of the device on the PCIe bus in BDF format. • [in] `domain` – IOMMU domain handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnIommuDetachDeviceFromDomain()`	Purpose Detaches a device on a PCIe bus from the defined IOMMU domain associated with the calling process. Parameters • [in] `bdf` – address of the device on the PCIe bus in BDF format. • [in] `domain` – IOMMU domain handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Using queues (queue.h)

The API is defined in the header file `sysroot-*-kos/include/kos/queue.h` from the KasperskyOS SDK.

The API sets up data exchange between threads owned by one process via a queuing mechanism that does not lock threads. In other words, you can add or extract elements of a queue without locking other threads that add or extract elements of this queue.

Information about API functions is provided in the table below.

Using the API

The standard scenario for API usage includes the following steps:

1. Create a queue abstraction.

A queue abstraction consists of a structure containing queue metadata and a queue buffer intended for storing elements of the queue. A queue buffer is logically divided into equal segments, each of which is intended for an individual element of the queue. The number of segments in a queue buffer matches the maximum number of elements in the queue. The alignment of segment addresses corresponds to the data types of elements in the queue.

To complete this step, call the `KosQueueCreate()` function. This function can allocate memory for the queue buffer or use already allocated memory. The size of the already allocated memory must be sufficient to accommodate the maximum number of elements in the queue. Also take into account that the size of a segment in the queue buffer is rounded to the next largest multiple of the alignment value defined through the `objAlign` parameter. The initial address of the already allocated memory must also be aligned to correspond to the data types of queue elements. If the memory address alignment specified in the `buffer` parameter is less than the value defined through the `objAlign` parameter, the function returns `RTL_NULL`.

2. Exchange data between threads by adding and extracting elements of the queue.

   To add one element to the end of the queue, you must reserve a segment in the queue buffer by calling the `KosQueueAlloc()` function, copy this element to the reserved segment, and call the `KosQueuePush()` function.

   To add a sequence of elements to the end of the queue, you must reserve the necessary number of segments in the queue buffer via `KosQueueAlloc()` function calls, copy the elements of this sequence to the reserved segments, and call the `KosQueuePushArray()` function. The order of elements in a sequence is not changed after this sequence is added to the queue. In other words, elements are added to the queue in the same order in which the pointers to reserved segments in the queue buffer are put into the array that is passed through the `objs` parameter of the `KosQueuePushArray()` function.

   To extract the first element of the queue, you must call the `KosQueuePop()` function. This function returns the pointer to the reserved segment in the queue buffer that contains the first element of the queue. After using an extracted element (for example, after checking or saving the value of an element), you must free the queue buffer segment occupied by this element. To do so, call the `KosQueueFree()` function.

   To check whether the queue buffer has no reserved segments, call the `KosQueueIsEmpty()` function.

   To clear the queue and free all registered segments in the queue buffer, you must call the `KosQueueFlush()` function.

3. Delete the queue abstraction.

   To complete this step, call the `KosQueueDestroy()` function. This function flushes the queue buffer if only the memory for this buffer was allocated by the `KosQueueCreate()` function. Otherwise, you must separately flush the queue buffer.

Information about API functions

queue.h functions

Function	Information about the function
`KosQueueCreate()`	Purpose Creates a queue abstraction. Parameters • [in] `objCount` – maximum number of elements in the queue. • [in] `objSize` – size (in bytes) of an element in the queue. • [in] `objAlign` – alignment of segment addresses in the queue buffer. The addresses of segments in the queue buffer may be unaligned (*objAlign=1*) or aligned (*objAlign=2,4,...,2^N*) to the boundary of a *2^N*-byte sequence (for example, two-byte or four-byte). • [in,optional] `buffer` – pointer to the allocated memory for the queue buffer, or `RTL_NULL` to automatically allocate the memory. Returned values If successful, the function returns the queue abstraction ID, otherwise it returns `RTL_NULL`.
`KosQueueDestroy()`	Purpose Deletes a queue abstraction. Parameters • [in] `queue` – queue abstraction ID. Returned values N/A
`KosQueueAlloc()`	Purpose Reserves a segment in the queue buffer. Parameters • [in] `queue` – queue abstraction ID. Returned values If successful, the function returns the pointer to the reserved segment in the queue buffer, otherwise it returns `RTL_NULL`.
`KosQueueFree()`	Purpose Resets the reservation of the defined segment in the queue buffer. Parameters • [in] `queue` – queue abstraction ID. • [in] `obj` – pointer to the reserved segment in the queue buffer. Returned values N/A

Function	Information about the function
`KosQueuePush()`	Purpose Adds an element to the end of the queue. Parameters • [in] `queue` – queue abstraction ID. • [in] `obj` – pointer to the reserved segment in the queue buffer. Returned values N/A
`KosQueuePushArray()`	Purpose Adds a sequence of elements to the end of the queue. Parameters • [in] `queue` – queue abstraction ID. • [in] `objs` – array of pointers to reserved segments in the queue buffer. • [in] `count` – number of elements in the sequence. Returned values N/A
`KosQueuePop()`	Purpose Extracts the first element of the queue. Parameters • [in] `queue` – queue abstraction ID. • [in] `timeout` – timeout (in milliseconds) for an element to appear in the queue, or `INFINITE_TIMEOUT` to set an unlimited timeout. The constant `INFINITE_TIMEOUT` is defined in the header file `sysroot-*-kos/include/rtl/rtc.h` from the KasperskyOS SDK. Returned values If successful, this function returns the pointer to the reserved segment in the queue buffer containing the first element of the queue. Otherwise it returns `RTL_NULL`. Additional information Non-blocking call if the `timeout` parameter is set to `0`.
`KosQueueFlush()`	Purpose Clears the queue and resets the reservation of all registered segments in the queue buffer. Parameters • [in] `queue` – queue abstraction ID. Returned values N/A

Function	Information about the function
`KosQueueIsEmpty()`	Purpose Checks whether the queue buffer has no reserved segments. Parameters • [in] `queue` – queue abstraction ID. Returned values If the verification is successful, the function returns `rtl_true`, otherwise it returns `rtl_false`.

Using memory barriers (barriers.h)

This API is defined in the header file `sysroot-*-kos/include/coresrv/io/barriers.h` from the KasperskyOS SDK.

The API sets barriers for reading from memory and/or writing to memory. A *memory barrier* is an instruction for a compiler or processor that guarantees that memory access operations specified in source code before setting a barrier will be executed before the memory access operations specified in source code after setting a barrier. Use of memory barriers is required if the specific order of memory write and memory read operations is important. Otherwise, the optimization mechanisms of a compiler and/or processor could cause these operations to be executed in a different order than the order specified in the source code.

Information about API functions is provided in the table below.

barriers.h functions

Function	Information about the function
`IoReadBarrier()`	Purpose Sets a barrier for reading from memory. Parameters N/A Returned values N/A
`IoWriteBarrier()`	Purpose Sets a barrier for writing to memory. Parameters N/A Returned values N/A
`IoReadWriteBarrier()`	Purpose Sets a barrier for writing to memory and reading from memory. Parameters N/A Returned values N/A

Executing system calls (syscalls.h)

This API is defined in the header file `sysroot-*-kos/include/coresrv/syscalls.h` from the KasperskyOS SDK.

The API allows execution of the `Call()`, `Recv()`, and `Reply()` system calls for sending and receiving IPC messages.

Information about API functions is provided in the table below.

Using the API

Pointers to buffers containing the constant part and arena of IPC messages are passed to API functions by using an IPC message header whose type is defined in the header file `sysroot-*-kos/include/ipc/if_rend.h` from the KasperskyOS SDK. Prior to API function calls, the headers of IPC messages must be bound to buffers containing the constant part and arena of IPC messages. To do so, use the `PackInMsg()` and `PackOutMsg()` functions that are declared in the header file `sysroot-*-kos/include/services/rtl/nk_msg.h` from the KasperskyOS SDK.

The `Call()`, `CallEx()`, `Recv()`, and `RecvEx()` functions lock execution of the calling thread while waiting for the system calls to complete. The `CallEx()` and `RecvEx()` functions let you define the timeout for completion of a system call. When this timeout is reached, an uncompleted system call is interrupted and the thread that is waiting for its completion resumes execution. A system call is also interrupted if an error occurs during its execution (such as an error due to termination of a server process). If a thread waiting on the completion of a system call is terminated externally, this system call is also interrupted. A system call executed by the `CallEx()` or `RecvEx()` function can be interrupted (for example, for correct termination of a process) by using the API ipc_api.h.

If a system call was interrupted using the API ipc_api.h, the `CallEx()` and `RecvEx()` functions return the error code `rcIpcInterrupt`. If IPC message transmission is prohibited by security mechanisms (such as the Kaspersky Security Module or a capability-based security mechanism implemented by the KasperskyOS kernel), the `Call()`, `CallEx()`, and `Reply()` functions return the error code `rcSecurityDisallow`.

Information about API functions

syscalls.h functions

Function	Information about the function
`Call()`	Purpose Executes the `Call()` system call with an unlimited timeout for its completion. Parameters • [in] `handle` – client IPC handle. • [in] `msgOut` – pointer to the header of IPC requests. • [in,out] `msgIn` – pointer to the header of IPC responses. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`CallEx()`	Purpose Executes the `Call()` system call with a defined timeout for its completion and the capability to interrupt its execution. Parameters • [in] `handle` – client IPC handle. • [in] `msgOut` – pointer to the header of IPC requests. • [in,out] `msgIn` – pointer to the header of IPC responses. • [in] `mdelay` – timeout (in milliseconds) for completion of a `Call()` system call, or `INFINITE_TIMEOUT` to define an unlimited timeout. • [in,optional] `syncHandle` – handle of the IPC synchronization object, or `INVALID_HANDLE` if an interrupt of the `Call()` system call is not required. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`Reply()`	Purpose Executes the `Reply()` system call. Parameters • [in] `handle` – server IPC handle. • [in] `msgOut` – pointer to the header of IPC responses. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`Recv()`	Purpose Executes the `Recv()` system call with an unlimited timeout for its completion. Parameters • [in] `handle` – server IPC handle. • [in,out] `msgIn` – pointer to the header of IPC requests. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`RecvEx()`	Purpose Executes the `Recv()` system call with a defined timeout for its completion and the capability to interrupt its execution. Parameters • [in] `handle` – server IPC handle. • [in,out] `msgIn` – pointer to the header of IPC requests. • [in] `mdelay` – timeout (in milliseconds) for completion of a `Recv()` system call, or `INFINITE_TIMEOUT` to define an unlimited timeout. • [in,optional] `syncHandle` – handle of the IPC synchronization object, or `INVALID_HANDLE` if an interrupt of the `Recv()` system call is not required. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Managing IPC (ipc_api.h)

This API is defined in the header file `sysroot-*-kos/include/coresrv/ipc/ipc_api.h` from the KasperskyOS SDK.

API capabilities:

- Interrupt the `Call()` and `Recv()` system calls if one or more process threads are locked while waiting for these system calls to complete.

  For example, you may need to interrupt these system calls to correctly terminate a process so that threads waiting for the completion of these system calls can resume execution.

- Define the `Reply()` system call termination handlers.

  In a `Reply()` system call termination handler, you can close a handle that was transferred to a client, for example.

Information about API functions is provided in the table below.

IPC interrupt

The API interrupts system calls in process threads that were locked after the `CallEx()` or `RecvEx()` function was called from the API syscalls.h if these functions were called while specifying the IPC synchronization object handle in the `syncHandle` parameter. The state of the IPC synchronization object determines whether the blocking system calls `Call()` and `Recv()` will be interrupted. To create an IPC synchronization object, you must call the `KnIpcCreateSyncObject()` or `KnIpcCreateSyncObjectEx()` function. The `KnIpcCreateSyncObject()` function creates an IPC synchronization object whose state can be changed only from the outside. The `KnIpcCreateSyncObjectEx()` function is an expanded version of the `KnIpcCreateSyncObject()` function that lets you create an IPC synchronization object, which, after the first interruption, automatically switches to a state in which the `Call()` and `Recv()` system calls are not interrupted. Immediately after creation, the IPC synchronization object has a state in which the `Call()` and `Recv()` system calls are not interrupted.

> The handle of an IPC synchronization object cannot be transferred to another process because the necessary flag for this operation is not set in the permissions mask of this handle.

The `KnIpcSetInterrupt()` function switches an IPC synchronization object to a state that allows interruption of the `Call()` and `Recv()` system calls in those process threads that have been locked after a `CallEx()` or `RecvEx()` function call specifying the handle of this IPC synchronization object in the `syncHandle` parameter. The `KnIpcClearInterrupt()` function switches the IPC synchronization object back to the reverse state.

When the `Call()` and `Recv()` system calls are interrupted, calls of the `CallEx()` or `RecvEx()` functions end with the `rcIpcInterrupt` error and locked threads resume execution. A system call can be interrupted only during certain stages of its execution. The `Call()` system call that is executed by the `CallEx()` function can be interrupted only when the server has not yet received a `Recv()` or `RecvEx()` function call used to receive an IPC message that was sent to the server when calling the `CallEx()` function. The `CallEx()` system call executed by the `RecvEx()` function can be interrupted only while waiting for an IPC request from a client.

An IPC synchronization object that was created by calling the `KnIpcCreateSyncObjectEx()` function with the `rtl_false` value in the `autoclear` parameter or by calling the `KnIpcCreateSyncObject()` function has the following characteristics:

- It does not change its state after interruption of the `Call()` or `Recv()` system calls, and therefore subsequent calls of the `CallEx()` and `RecvEx()` functions will end with the `rcIpcInterrupt` error until the `KnIpcClearInterrupt()` function is called.

- When calling the `KnIpcSetInterrupt()` function, if there are no threads that were locked after calling the `CallEx()` or `RecvEx()` function, subsequent calls of these functions will end with the `rcIpcInterrupt` error until the `KnIpcClearInterrupt()` function is called.

An IPC synchronization object that was created by calling the `KnIpcCreateSyncObjectEx()` function with the `rtl_true` value in the `autoclear` parameter has the following characteristics:

- It changes its state after interruption of the `Call()` or `Recv()` system calls, and therefore subsequent calls of the `CallEx()` and `RecvEx()` functions will not end with the `rcIpcInterrupt` error until the `KnIpcSetInterrupt()` function is called.

- When calling the `KnIpcSetInterrupt()` function, if there are no threads that were locked after calling the `CallEx()` or `RecvEx()` function, the first of subsequent calls of these functions will end with the `rcIpcInterrupt` error, after which the IPC synchronization object will switch to a state in which the `Call()` and `Recv()` system calls are not interrupted.

To delete an IPC synchronization object, you must close its handle.

Installing the Reply() system call termination handlers

To define the `Reply()` system call termination handler for a thread, call the `KnIpcInstallPostReplyHandler()` function from this thread. For one thread you can define several such handlers that will be run once immediately after this thread executes the `Reply()` system call. After that, the set of handlers will be cleared and remain empty until the next call of the `KnIpcInstallPostReplyHandler()` function.

Use the `callback` parameter of the `KnIpcInstallPostReplyHandler()` function to specify a callback function that does not return values and has two parameters. The `replyRc` input parameter contains the code with which the `Reply()` system call ended (`rcOk` if successful). The `context` input parameter contains the value specified in the identically named parameter of the `KnIpcInstallPostReplyHandler()` function.

Information about API functions

ipc_api.h functions

Function	Information about the function
`KnIpcCreateSyncObject()`	Purpose Creates an IPC synchronization object. Parameters • [out] `syncHandle` – pointer to the handle of the IPC synchronization object. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KnIpcCreateSyncObjectEx()`	Purpose Creates an IPC synchronization object. Parameters • [in] `autoclear` – boolean value that indicates whether or not you need to create an IPC synchronization object that automatically switches to a state in which the `Call()` and `Recv()` system calls are not interrupted after the first interruption of these system calls (`rtl_true` means yes and `rtl_false` means no). • [out] `syncHandle` – pointer to the handle of the IPC synchronization object. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnIpcSetInterrupt()`	Purpose Switches the defined IPC synchronization object to a state in which the `Call()` and `Recv()` system calls are interrupted. Parameters • [in] `syncHandle` – handle of the IPC synchronization object. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnIpcClearInterrupt()`	Purpose Switches the defined IPC synchronization object to a state in which the `Call()` and `Recv()` system calls are not interrupted. Parameters • [in] `syncHandle` – handle of the IPC synchronization object. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnIpcInstallPostReplyHandler()`	Purpose Defines a `Reply()` system call termination handler for the calling thread. Parameters • [in] `callback` – ID of the callback function that is called immediately after the `Reply()` system call is executed. • [in,optional] `context` – pointer to the data that will be passed to the callback function that was defined via the `callback` parameter. You can specify `RTL_NULL` if data does not need to be passed. The callback function will receive the passed data through its own `context` parameter. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Using sessions (session.h)

The API is defined in the header file `sysroot-*-kos/include/kos/session.h` from the KasperskyOS SDK.

The API is intended for use in the code of resource providers that provide access to user resources via IPC, and in the code of programs that utilize user resources locally (without IPC). The API lets you organize session-based management of access to user resources. A session consists of a sequence of actions, such as opening access to a resource, using the resource, and closing access to the resource. When access to a resource is opened, a session is created. Closing access to the resource terminates the session. Multiple operations can be simultaneously performed with a resource in one session. A resource can be used in exclusive mode or multi-access mode. In the latter case, the resource must be opened multiple times (possibly with different access permissions). In other words, the resource will be used via multiple, simultaneously existing sessions (parallel sessions).

In the API description, an "object" refers to a `KosObject`, which is an abstraction of a resource (for details about `KosObject`s, see Using KosObjects (objects.h)). The type of session and session context are also `KosObjects`. (A session context contains information about this session, such as the resource access permissions and the counter for active operations with the resource.)

Information about API functions is provided in the table below.

Creating a session type

A session type is created before sessions are created. To create a session type, call the `KosSessionCreateType()` function.

You must use the `objectType` parameter to define the type of objects for which the created type of sessions are intended. The object type must inherit the type with the ID `kosSessionBaseObjectType`.

Use the `sessionType` parameter to define the type of session contexts. You can specify the base type of session contexts with the ID `kosSessionBaseSessionType` or the type that inherits it. The base type of session contexts corresponds to the structure containing data such as the pointer to the object, the resource access permissions, the resource handle permissions mask that is transferred to resource consumers, and the counter of active operations with the resource during the session (the type of this structure is not exported to programs, but instead is used by the `libkos` library). You can add additional information to the session context by creating your own session context type that inherits the base type. For example, the session context for device usage may additionally contain pointers to functions that are called by solution programs to work with the device.

You must use the `ops` parameter to define the callback functions that are called when creating and terminating sessions. The callback functions for the `open` and `close` IDs must be defined. You do not have to define the callback function for the `cancel` ID, therefore this ID can be set to `RTL_NULL`.

The callback function corresponding to the `open` ID is called when creating a session and receives the pointers to the object, session context, data passed to the `KosSessionOpenObject()` function via the `openCtx` parameter, and the buffer for saving the resource handle permissions mask transferred to resource consumers. In this callback function, you can check the status of the resource, define the resource handle permissions mask transferred to resource consumers, and write data to the object or session context, for example. If the return code is different from `rcOk`, a session will not be created.

The callback function corresponding to the `cancel` ID is called at the end of the session, even if operations with the resource have not yet been completed (the counter of active operations may be greater than zero), and receives pointers to the session context and object. In this callback function, you need to complete operations with the resource in the terminating session, and you can write data to the session context or object, for example.

The callback function corresponding to the `close` ID is called when terminating a session only after completion of all operations with the resource (the counter of active operations in the terminating session is equal to zero) and receives pointers to the session context and object. In this callback function, you can write data to an object, for example.

Registering an object

After creating a session type, you must register the required objects as objects that can be linked to sessions of this type. To register an object, call the `KosSessionRegisterObject()` function.

You cannot register one object to be linked to different types of sessions at the same time.

After registration, the object is ready to be opened.

Opening an object

When an object is opened, a session linked to this object is created, and an object is opened when there is a request to open access to a resource. This may be a request via IPC from the resource consumer or a local request from another program component.

To open an object, call the `KosSessionOpenObject()` function. You must use the `requiredRights` parameter to define the resource access permissions in the created session, and use the `shareMode` parameter to define the multi-access permissions for a resource to properly restrict access to this resource for parallel sessions. For example, if read-and-write permissions are defined via the `requiredRights` parameter and the `shareMode` parameter is set to `0`, the created session requires exclusive read-and-write access to the resource. If read-and-write permissions are defined via the `requiredRights` parameter but only read permissions are defined via the `shareMode` parameter, the created session requires read-and-write permissions and allows read operations in parallel sessions. If read permissions are defined via the `requiredRights` parameter but read-and-write permissions are defined via the `shareMode` parameter, the created session requires read-access and allows read-and-write operations in parallel sessions.

If a resource needs to be used in multi-access mode, you must consider the following conditions when the object is opened a second time and subsequent times:

- The access permissions defined via the `requiredRights` parameter cannot exceed the minimal access permissions defined via the `shareMode` parameter for all parallel sessions linked to this object.
- The access permissions defined via the `shareMode` parameter must include all of the access permissions defined via the `requiredRights` parameter for all parallel sessions linked to this object.

The session context will be created after the object is opened.

If a program uses resources locally, after the object is opened, the program component managing the resources must pass the pointer of the created session context to the program component that uses the resources. When subsequently requesting operations with a resource, the program component using the resources must pass this pointer to the program component managing the resources for session identification.

After opening an object, the resource provider must transfer the resource handle to the consumer. The session context must be used as the resource transfer context so that the consumer can receive the handle of the resource linked to the session. When subsequently requesting operations with a resource, the consumer must put the received resource handle into IPC requests so that the provider receives the pointer to the session context after dereferencing this handle. To fill the resource handle transport container with data for transfer to a resource consumer, call the `KosSessionContextToIpcHandle()` function. This function receives the session context via the `sessionCtx` parameter and transfers the resource handle transport container via the `desc` parameter.

The `KosSessionContextToIpcHandle()` function takes the permissions mask defined in the callback function that is called when creating the session and puts this permissions mask into the resource handle transport container. The access permissions defined in this mask must not exceed the session access permissions defined via the `requiredRights` parameter of the `KosSessionOpenObject()` function. However, the same access permission may correspond to different bits in the permissions mask and in the value defined via the `requiredRights` parameter of the `KosSessionOpenObject()` function.

The `KosSessionContextToIpcHandle()` function defines the session context as the resource transfer context. The session context will be automatically deleted (and therefore the session will be terminated) after closure or revocation of the resource handles that were generated by the transfer of the resource handle that was received by calling the `KosSessionContextToIpcHandle()` function. If you need to prevent automatic termination of a session for subsequent termination via the `KosSessionCloseObject()` or `KosSessionCloseObjectByIpcHandle()` function call, prior to transferring the resource handle you must increment the reference counter for the context of this session by calling the `KosRefObject()` function from the objects.h API. In this case, after calling the `KosSessionCloseObject()` of `KosSessionCloseObjectByIpcHandle()` function, you must decrement the reference counter for the session context by calling the `KosPutObject()` function from the objects.h API.

Performing operations with a resource

An operation with a resource (for example, reading, writing, or getting parameters) is performed by request via IPC from the resource consumer or by local request from another program component.

The scenario for performing an operation with a resource includes the followings steps:

1. Verify the resource access permissions.

   When processing a local request, call the `KosSessionGetOpenRights()` function. This function gets information about session resource access permissions defined via the `requiredRights` parameter of the `KosSessionOpenObject()` function to compare these permissions with the permissions needed to perform the operation.

   When processing an IPC request, you need to call the `KosSessionIpcHandleToSession()` function, which gets the session context and verifies whether the permissions mask of the resource handle that the resource consumer put into the IPC request corresponds to the requested operation. (This function must be called between the `Recv()` and `Reply()` system calls.) The resource handle transport container received from the resource consumer must be passed to the function via the `desc` parameter. In the `operation` parameter, you must specify the resource access permissions that are needed to perform the requested operation. In the `type` parameter, you must specify the ID of the session type to verify that the type of session context matches the session type. If you set the `type` parameter to `RTL_NULL`, you must verify the type of session context prior to converting the pointer type that was received via the `sessionCtx` parameter. Verification of the session context type is mandatory because the resource provider may create sessions with contexts of different types while the resource consumer may erroneously make an IPC request with the resource handle matching a session with a different type of context. To verify the context type, use the objects.h API.

2. Increment the counter for active operations in the session by calling the `KosSessionRef()` function.

   This function verifies that the session has not terminated. In the `type` parameter, you must specify the ID of the session type to verify that the type of session context matches the session type. You can use the `object` parameter to receive a pointer to the object.

3. Perform the requested actions with the resource.

4. Decrement the counter for active operations in the session by calling the `KosSessionPut()` function.

Getting a resource handle

A resource handle may be required for using notifications or for querying the Kaspersky Security Module through the security interface, for example. To get a resource handle for an object or session context, call the `KosSessionGetObjectHandle()` or `KosSessionGetObjectHandleBySession()` function, respectively.

Traversing sessions linked to an object

You may need to traverse sessions linked to an object to notify each of the resource consumers using a resource about the state of that resource, for example. To traverse sessions linked to an object, call the `KosSessionWalk()` function. Use the `handler` parameter to define the callback function that is called for each session during traversal and receives pointers to the object, session context and data that is passed to the `KosSessionWalk()` function via the `walkCtx` parameter. In the callback function, you can send a notification about the state of a resource to consumers of that resource, for example. The callback function must not call the `KosSessionOpenObject()`, `KosSessionCloseObject()`, `KosSessionRegisterObject()`, and `KosSessionUnregisterObject()` functions. If the callback function returns `rtl_false`, the traversal is completed.

Closing an object

When an object is closed, one of the sessions linked to this object is terminated. A session will not be terminated until all operations in this session are terminated (in other words, when the counter of active operations is equal to zero). When a session is terminated, the handles of the resource linked to this session are automatically revoked.

An object is closed when there is a request to close access to the resource. This may be a request via IPC from the resource consumer or a local request from another program component. After making a request to close access to a resource, the resource consumer must close the handle of this resource. (If the resource consumer terminates without making a request to close access to the resource, or closes the resource handle without making this request, the corresponding session will be automatically terminated under the condition that the resource consumer has not additionally incremented the reference counter for the context of this session.)

To close an object, call the `KosSessionCloseObject()` or `KosSessionCloseObjectByIpcHandle()` function.

The `KosSessionCloseObject()` function terminates the session corresponding to the session context that was defined via the `sessionCtx` parameter.

The `KosSessionCloseObjectByIpcHandle()` function ends the session associated with the resource handle placed by the resource consumer into the IPC request demanding to close access to the resource. (This function must be called between the `Recv()` and `Reply()` system calls when processing an IPC request.) The received resource handle transport container must be passed to the function via the `desc` parameter. In the `type` parameter, you must specify the ID of the session type to verify that the type of session context matches the session type.

Deregistering an object

When an object is deregistered, all sessions linked to this object are terminated and it is no longer possible to open this object. A session will not be terminated until all operations in this session are terminated (in other words, when the counter of active operations is equal to zero). When a session is terminated, the handles of the resource linked to this session are automatically revoked. (Revocation does not close resource handles, therefore revoked handles need to be closed.)

To deregister an object, call the `KosSessionUnregisterObject()` function.

Deleting a session type

A session type must be deleted if there are no registered objects for which sessions of this type can be used, and there will be no more registrations. For example, a session type must be deleted when the resource provider is terminated.

To delete a session type, call the `KosSessionDestroyType()` function.

Information about API functions

session.h functions

Function	Information about the function
`KosSessionCreateType()`	Purpose Creates a session type. Parameters • [in] `objectType` – ID of the object type for which the sessions are intended. The parameter type is defined in the header file `sysroot-*-kos/include/kos/objects.h` from the KasperskyOS SDK. • [in] `sessionType` – ID of the session context type. The parameter type is defined in the header file `sysroot-*-kos/include/kos/objects.h` from the KasperskyOS SDK. • [in] `ops` – pointer to the structure containing the IDs of callback functions that are called when sessions are created and terminated. • [out] `outType` – pointer to the ID of the session type. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosSessionDestroyType()`	Purpose Deletes a session type. Parameters • [in] `type` – ID of the session type. Returned values N/A

Function	Information about the function
`KosSessionRegisterObject()`	Purpose Registers an object as an object that can be linked to sessions of the defined type. Parameters • [in] `type` – ID of the session type. • [in] `object` – pointer to the object. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosSessionUnregisterObject()`	Purpose Deregisters an object that was registered by calling the `KosSessionRegisterObject()` function. Parameters • [in] `object` – pointer to the object. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosSessionOpenObject()`	Purpose Opens an object. Parameters • [in] `object` – pointer to the object. • [in,optional] `openCtx` – pointer to the data that will be passed to the callback function that is called when the session is created. You can specify `RTL_NULL` if data does not need to be passed. The callback function will receive the passed data through its own `ctx` parameter. • [in] `requiredRights` – value that defines the resource access permissions in a created session. • [in] `shareMode` – value that defines multi-access permissions for a resource. • [out] `sessionCtx` – pointer to the session context address. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosSessionCloseObject()`	Purpose Closes an object. Parameters • [in] `sessionCtx` – pointer to the session context. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KosSessionRef()`	Purpose Increments the counter for active operations in a session. Parameters • [in,optional] `type` – ID of the session type, or `RTL_NULL` if there is no need to verify that the session context type matches the session type. • [in] `sessionCtx` – pointer to the session context. • [out,optional] `object` – pointer to the object address, or `RTL_NULL` if the object address is not required. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosSessionPut()`	Purpose Decrements the counter for active operations in a session. Parameters • [in] `sessionCtx` – pointer to the session context. Returned values N/A
`KosSessionWalk()`	Purpose Traverses sessions bound to a defined object. Parameters • [in] `object` – pointer to the object. • [in] `handler` – ID of the callback function that is called for each session during traversal. • [in,optional] `walkCtx` – pointer to the data that will be passed to the callback function that was defined via the `handler` parameter. You can specify `RTL_NULL` if data does not need to be passed. The callback function will receive the passed data through its own `walkCtx` parameter. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosSessionGetOpenRights()`	Purpose Gets information about resource access permissions in the specified session. These permissions are defined via the `requiredRights` parameter of the `KosSessionOpenObject()` function. Parameters • [in] `sessionCtx` – pointer to the session context. Returned values: If successful, the function returns the value indicating the resource access permissions, otherwise it returns `0`.

Function	Information about the function
`KosSessionContextToIpcHandle()`	Purpose Fills the resource handle transport container with data for transfer to a resource consumer when processing an IPC request to open access to the resource. Parameters • [in] `sessionCtx` – pointer to the session context. • [out] `desc` – pointer to the handle transport container. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosSessionIpcHandleToSession()`	Purpose Gets the session context when processing an IPC request from a resource consumer. Parameters • [in] `desc` – pointer to the handle transport container. • [in] `operation` – value that indicates the resource access permissions required for performing operations to verify that the handle permissions mask put into the IPC request by the resource consumer matches the requested operation. • [in,optional] `type` – ID of the session type, or `RTL_NULL` if there is no need to verify that the session context type matches the session type. • [out] `sessionCtx` – pointer to the session context address. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosSessionCloseObjectByIpcHandle()`	Purpose Closes an object when processing an IPC request from a resource consumer to close access to the resource. Parameters • [in] `desc` – pointer to the handle transport container. • [in,optional] `type` – ID of the session type, or `RTL_NULL` if there is no need to verify that the session context type matches the session type. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosSessionGetObjectHandle()`	Purpose Gets a resource handle for an object. Parameters • [in] `object` – pointer to the object. • [out] `handle` – pointer to the handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KosSessionGetObjectHandleBySession()`	Purpose Gets a resource handle for a session context. Parameters • [in] `sessionCtx` – pointer to the session context. • [out] `handle` – pointer to the handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Using KosObjects (objects.h)

The API is defined in the header file `sysroot-*-kos/include/kos/objects.h` from the KasperskyOS SDK.

The API lets you use typified `KosObjects`. A `KosObject` is a process memory region consisting of segments containing service data and payload data. Service data includes a pointer to the structure describing the object type, a reference counter for the object, and other information about the object. Payload data may consist of any data depending on the purpose of the object. For example, segments of payload data in an object representing a device abstraction save the structures that contain the device operating parameters, pointers to low-level methods for working with the device, and the parameters of software/hardware interfaces for device management.

A type inheritance mechanism is supported for `KosObjects`. For example, an abstraction of the NS16550 UART device contains the following structures in different segments of payload data: a structure specific to NS16550 UART devices, a structure specific to the UART device class, and common structures for all devices. The NS16550 UART device abstraction type corresponds only to the structure that is specific to NS16550 UART devices. All other structures correspond to types that inherited the NS16550 UART device abstraction type.

In the API, the pointer to an object is the pointer to the segment of payload data in this object.

Information about API functions is provided in the table below.

Creating an object type

An object type is created before objects are created. An object type is a process memory region containing the structure that includes data such as the type name, pointers to functions that are called when objects of this type are created and destroyed, a pointer to the parent type, and a reference counter for the type. Each object stores the pointer to this structure, which means that the objects are typified.

To create an object type, call the `KosCreateObjectType()` or `KosCreateObjectTypeEx()` function. In contrast to the `KosCreateObjectType()` function, the `KosCreateObjectTypeEx()` function lets you create a type that inherits other types. This type contains the address of the parent type that is defined via the `parentType` parameter. You can also use the `parentType` parameter to define a type that also contains the address of its parent type. As a result, you can use the `KosCreateObjectTypeEx()` function to create a type inheritance tree. Each type in this tree will inherit the entire chain of types that form the path from the root of the tree to this specific type.

Depending on whether type inheritance is being employed, objects may be simple or composite. Objects that correspond to one type are known as simple objects. Objects that correspond to a type inheritance chain are known as composite objects. Composite ones consist of simple objects. The number of simple objects in a composite object is equal to the number of types in the inheritance chain. The simple objects form an inheritance chain corresponding to the chain of inheritance of their types. In other words, each object except the first one in the inheritance chain contains the address of the parent object.

You need to use the `defaultObjSize` parameter of the `KosCreateObjectType()` and `KosCreateObjectTypeEx()` functions to define the default size of objects (you can define a different size when creating an object). When creating a type that inherits other types, this parameter pertains only to the last type in the inheritance chain. The `defaultObjSize` parameter effectively defines the minimum size of a segment of payload data in simple objects because the actual size of objects will be larger than the defined size due to the alignment of memory regions, inheritance of types, and the presence of segments containing service data.

You can use the `ops` parameter of the `KosCreateObjectType()` and `KosCreateObjectTypeEx()` functions to define the callback functions that are called when objects are created and destroyed (you can define both functions or just one of them). When a composite object is created, a set of simple objects is created within it. When the composite object is destroyed, the set of simple objects within it is also destroyed. When each simple object is created and destroyed, separate callback functions that were defined when creating the type of this object are called. The first object in the inheritance chain is created first, and the last object in the inheritance chain is created last. The last object in the inheritance chain is destroyed first, and the first object in the inheritance chain is destroyed last.

The callback function that is called when creating an object receives pointers to the object and the data passed to the `KosCreateObjectEx()` function via the `context` parameter. In this callback function, you can allocate resources and initialize data in an object, for example. If the return code differs from `rcOk`, the object will not be created. If a composite object is being created without creating at least one simple object, this composite object will not be created. Also, when creating a composite object, the data passed to the `KosCreateObjectEx()` function via the `context` parameter will be received only by the callback function that is called when creating the last simple object in the inheritance chain.

The callback function that is called when an object is destroyed receives the pointer to the object and does not return values. In this callback function, you can free up resources, for example.

Creating an object

After creating an object type, you can create objects. To create an object, you must call the `KosCreateObject()` or `KosCreateObjectEx()` function.

Use the `type` parameter to define the object type. If you define a type that does not inherit other types, a simple object will be created. If you define a type that inherits other types, a composite object will be created and the last simple object in the inheritance chain for this composite object will have the defined type.

Use the `size` parameter to define the object size, or specify `0` to set the default object size specified in the type of this object. If a composite object is created, this parameter pertains only to the last simple object in the inheritance chain. The `size` parameter effectively defines the minimum size of a segment of payload data in a simple object because the actual size of the object will be larger than the defined size due to the alignment of the memory region, inheritance of types, and the presence of segments containing service data.

In contrast to the `KosCreateObject()` function, the `KosCreateObjectEx()` function has the `context` parameter, which can be used to pass data to the callback function that is called when an object is created. When creating a composite object, the data passed via the `context` parameter will be received only by the callback function that is called when creating the last simple object in the inheritance chain.

The `outObject` parameter is used by the `KosCreateObject()` and `KosCreateObjectEx()` functions to transfer the pointer of the created object. The pointer to the created composite object is the pointer to the last simple object in the inheritance chain.

After it is created, the object can be used to write data to it and read data from it.

Obtaining access to simple objects in a composite object

To gain access to simple objects in a composite object, you need to use the `KosGetObjectParent()` function and/or the `KosGetObjectAncestor()` function. The `KosGetObjectParent()` function gets the pointer to the object that is the direct ancestor of the defined object. The `KosGetObjectAncestor()` function gets the pointer to the object that is the ancestor of the defined object and has the defined type.

Managing the lifetime of objects and their types

An object exists as long as the reference counter for that object has a value greater than zero. Likewise, an object type exists as long as the reference counter for that object has a value greater than zero. When an object or object type is created, the value of the links counter is equal to 1. This counter can then be incremented or decremented to manage the lifetime of the object or object type. For example, after an object is created, you can decrement the reference counter for its type because the `libkos` library increments the reference counter for this object type when an object is created. In this case, the object type will be destroyed automatically after the object is destroyed because the `libkos` library decrements the reference counter for the object type when an object is destroyed. In addition, when saving an object address in another object, you must increment the reference counter for the first object to ensure that it exists throughout the lifetime of the second object. When you destroy an object that contains the address of another object, you must decrement the reference counter for the second object to ensure that it is destroyed when no other links remain.

The lifetime of a simple object within a composite object corresponds to the lifetime of this composite object. You cannot separately manage the lifetime of a simple object within a composite object.

To increment the reference counter for an object or object type, call the `KosRefObject()` or `KosRefObjectType()` function, respectively.

To decrement the reference counter for an object or object type, call the `KosPutObject()` or `KosPutObjectType()` function, respectively.

Verifying an object type

The `KosCheckParentType()` and `KosObjectTypeInheritsType()` functions let you check whether one object type is a parent to another object type.

The `KosObjectOfType()` function verifies whether an object has the defined type.

The `KosObjectInheritsType()` function checks whether an object has the defined type or a type inherited by the defined type.

Getting the type and name of an object

To get the object type, call the `KosObjectGetType()` function. Increments the reference counter for an object type. If the received object type ID is no longer needed, you must decrement the reference counter for the object type by calling the `KosPutObjectType()` function.

To get the object name, call the `KosGetObjectName()` function. (Each simple object in a composite object has the name of this composite object.) The function increments the reference counter for the `KosString` containing the object name. If this string is no longer needed, you need to decrement the reference counter for it by calling the `KosPutString()` function from the strings.h API.

Getting the name of an object type

To get the name of an object type, call the `KosObjectGetTypeName()` function. Increments the reference counter for the `KosString` containing the object type name. If this string is no longer needed, you need to decrement the reference counter for it by calling the `KosPutString()` function from the strings.h API.

Information about API functions

objects.h functions

Function	Information about the function
`KosCreateObjectType()`	Purpose Creates an object type. Parameters • [in] `name` – pointer to the name of the object type. • [in] `defaultObjSize` – default size of objects (in bytes). • [in,optional] `ops` – pointer to the structure containing pointers to the callback functions that are called when creating and destroying objects. You can set this to `RTL_NULL` if you do not need to define these callback functions. • [out] `outType` – pointer to the ID of the object type. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KosCreateObjectTypeEx()`	Purpose Creates an object type. Parameters • [in] `name` – pointer to the name of the object type. • [in] `id` – fictitious parameter. You must specify `0`. • [in] `defaultObjSize` – default size of objects (in bytes). • [in,optional] `ops` – pointer to the structure containing pointers to the callback functions that are called when creating and destroying objects. You can set this to `RTL_NULL` if you do not need to define these callback functions. • [in,optional] `parentType` – ID of the parent object type, or `RTL_NULL` if inheritance of other object types is not required. • [out] `outType` – pointer to the ID of the object type. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosCreateObjectEx()`	Purpose Creates an object. Parameters • [in] `type` – ID of the object type. • [in,optional] `name` – pointer to the object name, or `RTL_NULL` if you do not need to define the object name. • [in,optional] `size` – size of the object (in bytes), or `0` to set the default object size. • [in,optional] `context` – pointer to the data that will be passed to the callback function that is called when the object is created. You can specify `RTL_NULL` if data does not need to be passed. The callback function will receive the passed data through its own `context` parameter. • [out] `outObject` – pointer to the object address. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information When an object is created, memory is allocated for that object and this memory is filled with zeros.
`KosRefObjectType()`	Purpose Increments the reference counter for an object type. Parameters • [in] `type` – ID of the object type. Returned values N/A

Function	Information about the function
`KosPutObjectType()`	Purpose Decrements the reference counter for an object type. Parameters • [in] `type` – ID of the object type. Returned values N/A
`KosCheckParentType()`	Purpose Verifies that the specific object `type` is a direct descendant of the `parentType` object type. Parameters • [in] `type` – ID of the object type. • [in] `parentType` – ID of the object type. Returned values: If the verification is successful, the function returns `rtl_true`, otherwise it returns `rtl_false`.
`KosObjectTypeInheritsType()`	Purpose Verifies that the specific object `type` is a descendant of the `parentType` object type. Parameters • [in] `type` – ID of the object type. • [in] `parentType` – ID of the object type. Returned values: If the verification is successful, the function returns `rtl_true`, otherwise it returns `rtl_false`.
`KosCreateObject()`	Purpose Creates an object. Parameters • [in] `type` – ID of the object type. • [in,optional] `name` – pointer to the object name, or `RTL_NULL` if you do not need to define the object name. • [in,optional] `size` – size of the object (in bytes), or `0` to set the default object size. • [out] `outObject` – pointer to the object address. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information When an object is created, memory is allocated for that object and this memory is filled with zeros.

Function	Information about the function
`KosRefObject()`	Purpose Increments the reference counter for an object. Parameters • [in] `object` – pointer to the object. Returned values N/A
`KosPutObject()`	Purpose Decrements the reference counter for an object. Parameters • [in] `object` – pointer to the object. Returned values N/A
`KosGetObjectName()`	Purpose Gets the object name. Parameters • [in] `object` – pointer to the object. Returned values: Returns the ID of the `KosString` containing the object name. The type of returned value is defined in the header file `sysroot-*-kos/include/kos/strings.h` from the KasperskyOS SDK.
`KosGetObjectParent()`	Purpose Gets the pointer to the object that is the direct ancestor of the defined object. Parameters • [in] `object` – pointer to the object. Returned values: If successful, the function returns the pointer for the object that is the direct ancestor of the defined object, otherwise it returns `RTL_NULL`.
`KosGetObjectAncestor()`	Purpose Gets the pointer to the object that is the ancestor of the defined object and has the defined type. Parameters • [in] `object` – pointer to the object. • [in] `type` – ID of the object type. Returned values: If successful, the function returns the pointer for the object that is the ancestor of the defined object and has the defined type, otherwise it returns `RTL_NULL`. Additional information If the `object` parameter is used to define an object that has the type defined via the `type` parameter, the function returns the pointer for this object.

Function	Information about the function
`KosObjectOfType()`	Purpose Verifies that the object has the defined type. Parameters • [in] `object` – pointer to the object. • [in] `type` – ID of the object type. Returned values: If the verification is successful, the function returns `rtl_true`, otherwise it returns `rtl_false`.
`KosObjectInheritsType()`	Purpose Verifies that the object has the defined type or the type that inherits the defined type. Parameters • [in] `object` – pointer to the object. • [in] `type` – ID of the object type. Returned values: If the verification is successful, the function returns `rtl_true`, otherwise it returns `rtl_false`.
`KosObjectGetType()`	Purpose Gets the ID of the object type. Parameters • [in] `object` – pointer to the object. Returned values: Object type ID.
`KosObjectGetTypeName()`	Purpose Gets the name of the object type. Parameters • [in] `type` – ID of the object type. Returned values: Returns the ID of the `KosString` containing the object type name. The type of returned value is defined in the header file `sysroot-*-kos/include/kos/strings.h` from the KasperskyOS SDK.

Using KosObject containers (objcontainer.h, objcontainernolock.h, basecontainer.h)

The APIs let you merge `KosObject`s into containers to conveniently use sets of these objects (for details about `KosObject`s, see Using KosObjects (objects.h)). The containers are also `KosObjects` and may be elements of other containers. The same `KosObject` may be an element of multiple containers at the same time.

Each API is intended for the use of various types of containers. The specific characteristics of each type of container are presented in the table below.

Specific characteristics of container types

Type name	Type characteristics
`KosObjContainer`	• Objects have names within a container. • The same object may be added to a container multiple times with different names. • Operations performed with objects are thread-safe.
`KosObjContainerNolock`	• Objects have names within a container. • The same object may be added to a container multiple times with different names. • Operations performed with objects are not thread-safe.
`KosBaseContainer`	• Objects do not have names within a container. • The same object cannot be added to a container multiple times. • Operations performed with objects are not thread-safe.

Using KosObjContainers (objcontainer.h)

The API is defined in the header file `sysroot-*-kos/include/kos/objcontainer.h` from the KasperskyOS SDK.

Information about API functions is provided in the table below.

Creating a container

To create a container, call the `KosObjContainerCreate()` function. In the `parent` parameter, you can specify the ID of the parent container, which is the `KosObjContainer`, `KosObjContainerNolock` or `KosBaseContainer` to which the created container will be added.

Adding an object to a container

To add an object to a container, call the `KosObjContainerInsert()` function. You can add any `KosObject`, including a `KosObjContainer`, `KosObjContainerNolock` or `KosBaseContainer`. You must use the `name` parameter to define the object name that this object will have within the container. This name is not related to the name that was defined when the object was created. The name of an object within a container must be unique. You can add the same object multiple times with different names. When an object is added to a container, the reference counter for this object is incremented.

Removing an object from a container

To remove an object from a container, call the `KosObjContainerRemoveByName()` or `KosObjContainerRemove()` function. When an object is removed from a container, the reference counter for this object is decremented.

Searching for an object in a container

To search for an object with the defined name in a container, call the `KosObjContainerGetByName()` function. Child containers will not be searched for the object. This function increments the reference counter for the found object.

Traversing objects in a container

To traverse objects in a container, call the `KosObjContainerWalk()` function. Use the `walk` parameter to define the callback function that is called for each object during traversal and receives pointers to the object and data that is passed to the `KosObjContainerWalk()` function via the `context` parameter. The callback function must not call other API functions. If the callback function returns a value with the `KOS_OBJECT_WALK_FINISH` flag, the `KosObjContainerWalk()` function stops traversal. If a callback function returns a value with the `KOS_OBJECT_WALK_REMOVE_OBJECT` flag, the `KosObjContainerWalk()` function removes the corresponding object from the container. (The flags are defined in the header file `sysroot-*-kos/include/kos/objects.h` from the KasperskyOS SDK.)

Objects in child containers are not traversed.

Enumerating the names of objects in a container

To enumerate the names of objects in a container, use the `KosObjContainerEnumName()` function. The order of enumeration for object names matches the order in which these objects are added to the container. Object names in child containers are not enumerated. The function increments the reference counter for the `KosString` containing the object name. If this string is no longer needed, you need to decrement the reference counter for it by calling the `KosPutString()` function from the strings.h API.

Getting the number of objects in a container

To get the number of objects in a container, call the `KosObjContainerCount()` function. Objects in child containers are not taken into account.

Clearing a container

To remove all objects from a container, call the `KosObjContainerClear()` function. This function decrements the counters for links to objects that are removed from the container.

Deleting a container

To delete a container, call the `KosObjContainerDestroy()` function. When a container is deleted, the counters for links to objects within this container are decremented.

Information about API functions

objcontainer.h functions

Function	Information about the function
`KosObjContainerCreate()`	Purpose Creates a container. Parameters • [in,optional] `parent` – ID of the parent container, or `RTL_NULL` if you do not need to add the created container to an already existing container. • [in] `name` – pointer to the name of the created container. The maximum size (bytes) of a container name matches the value of the `KOS_OBJ_CONTAINER_NAME_MAX` constant. • [out] `outContainer` – pointer to the ID of the created container. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosObjContainerDestroy()`	Purpose Deletes a container. Parameters • [in] `container` – ID of the container. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KosObjContainerInsert()`	Purpose Adds an object to a container. Parameters • [in] `container` – ID of the container. • [in] `name` – pointer to the object name. • [in] `object` – pointer to the object. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosObjContainerRemoveByName()`	Purpose Removes the object with the defined name from a container. Parameters • [in] `container` – ID of the container. • [in] `name` – pointer to the object name. • [in,optional] `object` – pointer to the object, or `RTL_NULL` if you do not need to verify that the object name defined via the `name` parameter matches the object being removed. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosObjContainerRemove()`	Purpose Removes an object from a container. Parameters • [in] `container` – ID of the container. • [in] `object` – pointer to the object. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosObjContainerGetByName()`	Purpose Searches a container for the object with the defined name. Parameters • [in] `container` – ID of the container. • [in] `name` – pointer to the name of the object that you are searching for. • [out] `outObject` – pointer to the address of the found object. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information If an object with the defined name is not found, the function returns the `RTL_NULL` value via the `outObject` parameter.

Function	Information about the function
`KosObjContainerWalk()`	Purpose Traverses objects in a container and calls the defined function for each object during traversal. Parameters • [in] `container` – ID of the container. • [in] `walk` – ID of the callback function that is called for each object during traversal. The parameter type is defined in the header file `sysroot-*-kos/include/kos/objects.h` from the KasperskyOS SDK. • [in,optional] `context` – pointer to the data that will be passed to the callback function that was defined via the `walk` parameter. You can specify `RTL_NULL` if data does not need to be passed. The callback function will receive the passed data through its own `context` parameter. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosObjContainerEnumName()`	Purpose Enumerates the names of objects in a container. Parameters • [in] `container` – ID of the container. • [in] `index` – index for enumerating object names. Enumeration starts with zero. • [out] `outStr` – pointer to the ID of the `KosString` containing the object name. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosObjContainerCount()`	Purpose Gets the number of objects in a container. Parameters • [in] `container` – ID of the container. Returned values: Number of objects in a container.
`KosObjContainerClear()`	Purpose Clears a container. Parameters • [in] `container` – ID of the container. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Using KosObjContainerNolock (objcontainernolock.h)

The API is defined in the header file `sysroot-*-kos/include/kos/objcontainernolock.h` from the KasperskyOS SDK.

Information about API functions and macros is provided in the table below.

Creating a container

To create a container, call the `KosObjContainerNolockCreate()` function. In the `parent` parameter, you can specify the ID of the parent container, which is the `KosObjContainer`, `KosObjContainerNolock` or `KosBaseContainer` to which the created container will be added.

Adding an object to a container

To add an object to a container, call the `KosObjContainerNolockInsert()` function. You can add any `KosObject`, including a `KosObjContainer`, `KosObjContainerNolock` or `KosBaseContainer`. You must use the `name` parameter to define the object name that this object will have within the container. This name is not related to the name that was defined when the object was created. The name of an object within a container must be unique. You can add the same object multiple times with different names. When an object is added to a container, the reference counter for this object is incremented.

Removing an object from a container

To remove an object from a container, call the `KosObjContainerNolockRemoveByName()` or `KosObjContainerNolockRemove()` function. When an object is removed from a container, the reference counter for this object is decremented.

Searching for an object in a container

To search for an object with the defined name in a container, call the `KosObjContainerNolockGetByName()` function. Child containers will not be searched for the object. This function increments the reference counter for the found object.

Navigation within a container

To initialize an iterator for navigation within a container and to receive access to objects, call the `KosObjContainerNolockIteratorInit()` function. After initialization, the iterator is in the initial position.

To move the iterator to the next position, call the `KosObjContainerNolockIteratorNext()` function.

To check whether the iterator is in the final position, call the `KosObjContainerNolockIteratorIsEnd()` function.

To get the object corresponding to the current position of the iterator, call the `KosObjContainerNolockIteratorGetObject()` function.

Traversing objects in a container

To traverse objects in a container, use the `KosObjContainerNolockWalk()` function or the `KOS_OBJ_CONTAINER_NOLOCK_FOR_EACH()` macro.

When using the `KosObjContainerNolockWalk()` function, use the `walk` parameter to define the callback function that is called for each object during traversal and receives pointers to the object and data that is passed to the `KosObjContainerNolockWalk()` function via the `context` parameter. If the callback function returns a value with the `KOS_OBJECT_WALK_FINISH` flag, the `KosObjContainerNolockWalk()` function stops traversal. If a callback function returns a value with the `KOS_OBJECT_WALK_REMOVE_OBJECT` flag, the `KosObjContainerNolockWalk()` function removes the corresponding object from the container. (The flags are defined in the header file `sysroot-*-kos/include/kos/objects.h` from the KasperskyOS SDK.)

The `KOS_OBJ_CONTAINER_NOLOCK_FOR_EACH()` macro is expanded into the header of the `for` operator. The pointer defined in the `obj` parameter will contain the address of the next object in each iteration. You can use this pointer to perform operations with objects. For example, you can delete an object from a container by calling the `KosObjContainerNolockRemove()` function.

Objects in child containers are not traversed.

Enumerating the names of objects in a container

To enumerate the names of objects in a container, use the `KosObjContainerNolockEnumName()` function. The order of enumeration for object names matches the order in which these objects are added to the container. Object names in child containers are not enumerated. The function increments the reference counter for the `KosString` containing the object name. If this string is no longer needed, you need to decrement the reference counter for it by calling the `KosPutString()` function from the strings.h API.

Getting the number of objects in a container

To get the number of objects in a container, call the `KosObjContainerNolockCount()` function. Objects in child containers are not taken into account.

Clearing a container

To remove all objects from a container, call the `KosObjContainerNolockClear()` function. This function decrements the counters for links to objects that are removed from the container.

Deleting a container

To delete a container, call the `KosObjContainerNolockDestroy()` function. When a container is deleted, the counters for links to objects within this container are decremented.

Information about API functions and macros

objcontainernolock.h functions and macros

Function/Macro	Information about the function/macro
`KosObjContainerNolockCreate()`	Purpose Creates a container. Parameters • [in,optional] `parent` – ID of the parent container, or `RTL_NULL` if you do not need to add the created container to an already existing container. • [in] `name` – pointer to the name of the created container. The maximum size (bytes) of a container name matches the value of the `KOS_OBJ_CONTAINER_UNSAFE_NAME_MAX` constant. • [out] `outContainer` – pointer to the ID of the created container. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosObjContainerNolockDestroy()`	Purpose Deletes a container. Parameters • [in] `container` – ID of the container. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosObjContainerNolockClear()`	Purpose Clears a container. Parameters • [in] `container` – ID of the container. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosObjContainerNolockInsert()`	Purpose Adds an object to a container. Parameters • [in] `container` – ID of the container. • [in] `name` – pointer to the object name. • [in] `object` – pointer to the object. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function/Macro	Information about the function/macro
`KosObjContainerNolockRemoveByName()`	Purpose Removes the object with the defined name from a container. Parameters • [in] `container` – ID of the container. • [in] `name` – pointer to the object name. • [in,optional] `object` – pointer to the object, or `RTL_NULL` if you do not need to verify that the object name defined via the `name` parameter matches the object being removed. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosObjContainerNolockRemove()`	Purpose Removes an object from a container. Parameters • [in] `container` – ID of the container. • [in] `object` – pointer to the object. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosObjContainerNolockGetByName()`	Purpose Searches a container for the object with the defined name. Parameters • [in] `container` – ID of the container. • [in] `name` – pointer to the name of the object that you are searching for. • [out] `outObject` – pointer to the address of the found object. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information If an object with the defined name is not found, the function returns the `RTL_NULL` value via the `outObject` parameter.

Function/Macro	Information about the function/macro
`KosObjContainerNolockWalk()`	Purpose Traverses objects in a container and calls the defined function for each object during traversal. Parameters • [in] `container` – ID of the container. • [in] `walk` – ID of the callback function that is called for each object during traversal. The parameter type is defined in the header file `sysroot-*-kos/include/kos/objects.h` from the KasperskyOS SDK. • [in,optional] `context` – pointer to the data that will be passed to the callback function that was defined via the `walk` parameter. You can specify `RTL_NULL` if data does not need to be passed. The callback function will receive the passed data through its own `context` parameter. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosObjContainerNolockEnumName()`	Purpose Enumerates the names of objects in a container. Parameters • [in] `container` – ID of the container. • [in] `index` – index for enumerating object names. Enumeration starts with zero. • [out] `outStr` – pointer to the ID of the `KosString` containing the object name. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosObjContainerNolockCount()`	Purpose Gets the number of objects in a container. Parameters • [in] `container` – ID of the container. Returned values: Number of objects in a container.
`KosObjContainerNolockIteratorInit()`	Purpose Initializes an iterator for navigation within a container and receives access to objects. Parameters • [in] `container` – ID of the container. Returned values: Initialized iterator for navigation within a container and access to objects.

Function/Macro	Information about the function/macro
`KosObjContainerNolockIteratorIsEnd()`	Purpose Checks whether the iterator is in the final position. Parameters • [in] `iterator` – pointer to the iterator. Returned values: If the verification is successful, the function returns `rtl_true`, otherwise it returns `rtl_false`.
`KosObjContainerNolockIteratorGetObject()`	Purpose Gets the object corresponding to the current position of the iterator. Parameters • [in] `iterator` – pointer to the iterator. Returned values: Pointer to the object. If the iterator is in the final position, the function returns `RTL_NULL`.
`KosObjContainerNolockIteratorNext()`	Purpose Moves the iterator to the next position. Parameters • [in] `iterator` – pointer to the iterator. Returned values N/A
`KOS_OBJ_CONTAINER_NOLOCK_FOR_EACH()`	Purpose Traverses objects in a container. Parameters • [in] `container` – ID of the container. • [in] `obj` – pointer for saving the addresses of traversed objects. Macro values N/A

Using KosBaseContainers (basecontainer.h)

The API is defined in the header file `sysroot-*-kos/include/kos/basecontainer.h` from the KasperskyOS SDK.

Information about API functions and macros is provided in the table below.

Creating a container

To create a container, call the `KosBaseContainerCreate()` function. In the `parent` parameter, you can specify the ID of the parent container, which is the `KosObjContainer`, `KosObjContainerNolock` or `KosBaseContainer` to which the created container will be added.

Adding an object to a container

To add an object to a container, call the `KosBaseContainerInsert()` function. You can add any `KosObject`, including a `KosObjContainer`, `KosObjContainerNolock` or `KosBaseContainer`. You cannot add the same object multiple times. When an object is added to a container, the reference counter for this object is incremented.

Adding objects to a container from another container

To add objects to a container from another container, call the `KosBaseContainerAppend()` function. The source container does not change.

Removing an object from a container

To remove an object from a container, call the `KosBaseContainerRemove()` function. When an object is removed from a container, the reference counter for this object is decremented.

Moving an object to another container

To move an object from one container to another container, call the `KosBaseContainerMove()` function.

Navigation within a container

To initialize an iterator for navigation within a container and to gain access to objects, call the `KosBaseContainerIteratorInit()` function. After initialization, the iterator is in the initial position.

To move the iterator to the next position, call the `KosBaseContainerIteratorNext()` function.

To check whether the iterator is in the final position, call the `KosBaseContainerIteratorIsEnd()` function.

To get the object corresponding to the current position of the iterator, call the `KosBaseContainerIteratorGetObject()` function.

Traversing objects in a container

To traverse objects in a container, use the `KosBaseContainerWalk()` function or the `KOS_BASE_CONTAINER_FOR_EACH()` macro.

When using the `KosBaseContainerWalk()` function, use the `walk` parameter to define the callback function that is called for each object during traversal and receives pointers to the object and data that is passed to the `KosBaseContainerWalk()` function via the `context` parameter. If the callback function returns a value with the `KOS_OBJECT_WALK_FINISH` flag, the `KosBaseContainerWalk()` function stops traversal. If a callback function returns a value with the `KOS_OBJECT_WALK_REMOVE_OBJECT` flag, the `KosBaseContainerWalk()` function removes the corresponding object from the container. (The flags are defined in the header file `sysroot-*-kos/include/kos/objects.h` from the KasperskyOS SDK.)

The `KOS_BASE_CONTAINER_FOR_EACH()` macro is expanded into the header of the `for` operator. The pointer defined in the `obj` parameter will contain the address of the next object in each iteration. You can use this pointer to perform operations with objects. For example, you can delete an object from a container by calling the `KosBaseContainerRemove()` function.

Objects in child containers are not traversed.

Enumerating objects in a container

To enumerate objects in a container, call the `KosBaseContainerEnum()` function. The order of object enumeration matches the order in which these objects are added to the container. Objects in child containers are not enumerated. This function increments the reference counter for the object. If this object is no longer needed, you need to decrement the reference counter by calling the `KosPutObject()` function from the objects.h API.

Getting the number of objects in a container

To get the number of objects in a container, call the `KosBaseContainerCount()` function. Objects in child containers are not taken into account.

Clearing a container

To remove all objects from a container, call the `KosBaseContainerClear()` function. This function decrements the counters for links to objects that are removed from the container.

Deleting a container

To delete a container, call the `KosBaseContainerDestroy()` function. When a container is deleted, the counters for links to objects within this container are decremented.

Information about API functions and macros

basecontainer.h functions and macros

Function/Macro	Information about the function/macro
`KosBaseContainerCreate()`	Purpose Creates a container. Parameters • [in,optional] `parent` – ID of the parent container, or `RTL_NULL` if you do not need to add the created container to an already existing container. • [out] `outContainer` – pointer to the ID of the created container. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosBaseContainerDestroy()`	Purpose Deletes a container. Parameters • [in] `container` – ID of the container. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosBaseContainerInsert()`	Purpose Adds an object to a container. Parameters • [in] `container` – ID of the container. • [in] `object` – pointer to the object. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosBaseContainerRemove()`	Purpose Removes an object from a container. Parameters • [in] `container` – ID of the container. • [in] `object` – pointer to the object. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function/Macro	Information about the function/macro
`KosBaseContainerMove()`	Purpose Moves an object from one container into another. Parameters • [in] `destination` – ID of the receiving container. • [in] `source` – ID of the source container. • [in] `object` – pointer to the object. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosBaseContainerWalk()`	Purpose Traverses objects in a container and calls the defined function for each object during traversal. Parameters • [in] `container` – ID of the container. • [in] `walk` – ID of the callback function that is called for each object during traversal. The parameter type is defined in the header file `sysroot-*-kos/include/kos/objects.h` from the KasperskyOS SDK. • [in,optional] `context` – pointer to the data that will be passed to the callback function that was defined via the `walk` parameter. You can specify `RTL_NULL` if data does not need to be passed. The callback function will receive the passed data through its own `context` parameter. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosBaseContainerEnum()`	Purpose Enumerates objects in a container. Parameters • [in] `container` – ID of the container. • [in] `index` – index for enumerating objects. Enumeration starts with zero. Returned values: Pointer to the object. If the index value is exceeded, the function returns `RTL_NULL`.
`KosBaseContainerCount()`	Purpose Gets the number of objects in a container. Parameters • [in] `container` – ID of the container. Returned values: Number of objects in a container.

Function/Macro	Information about the function/macro
`KosBaseContainerClear()`	Purpose Clears a container. Parameters • [in] `container` – ID of the container. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosBaseContainerAppend()`	Purpose Adds objects to a container from another container. Parameters • [in] `destination` – ID of the receiving container. • [in] `source` – ID of the source container. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosBaseContainerIteratorInit()`	Purpose Initializes an iterator for navigation within a container and receives access to objects. Parameters • [in] `container` – ID of the container. Returned values: Initialized iterator for navigation within a container and access to objects.
`KosBaseContainerIteratorIsEnd()`	Purpose Checks whether the iterator is in the final position. Parameters • [in] `iterator` – pointer to the iterator. Returned values: If the verification is successful, the function returns `rtl_true`, otherwise it returns `rtl_false`.
`KosBaseContainerIteratorGetObject()`	Purpose Gets the object corresponding to the current position of the iterator. Parameters • [in] `iterator` – pointer to the iterator. Returned values: Pointer to the object. If the iterator is in the final position, the function returns `RTL_NULL`.

Function/Macro	Information about the function/macro
`KosBaseContainerIteratorNext()`	Purpose Moves the iterator to the next position. Parameters • [in] `iterator` – pointer to the iterator. Returned values N/A
`KOS_BASE_CONTAINER_FOR_EACH()`	Purpose Traverses objects in a container. Parameters • [in] `container` – ID of the container. • [in] `obj` – pointer for saving the addresses of traversed objects. Macro values N/A

Using KosObject identifiers (objhandles.h)

The API is defined in the header file `sysroot-*-kos/include/kos/objhandles.h` from the KasperskyOS SDK.

The API lets you create tables of numerical identifiers related to `KosObject`s (for details about `KosObject`, see Using KosObjects (objects.h)). By using this table, you can verify the validity of an ID before converting it into a pointer to a `KosObject`. For example, a `KosObject` was created and the pointer to this object was converted into a numerical identifier that was later corrupted. To perform operations with the `KosObject`, this identifier must be converted into a pointer. Before this conversion, a search is performed for the identifier in the table, and therefore it is highly likely that the invalidity of the identifier will be detected and an error will be received. If a direct pointer were used in this example to identify the `KosObject`, a corruption of this pointer could lead to unforeseen consequences, potentially without receiving an error.

Information about API functions is provided in the table below.

Creating an object identifier table

To create an object identifier table, call the `KosCreateObjHandleTable()` function. Creates a table containing the number of identifiers defined in the `initialCount` parameter. If this number turns out to be insufficient during subsequent use of the table, additional identifiers will be automatically added. The maximum number of identifiers (including those added after creating the table) is defined in the `maxCount` parameter.

Binding an object to an identifier in the table

To bind an object to an identifier in the table, call the `KosCreateObjHandle()` function. This function increments the reference counter for the object.

Getting a pointer to an object based on an identifier in the table

To get a pointer to an object based on an identifier in the table, call the `KosRefObjectByObjHandle()` function. This function increments the reference counter for the object. If the received pointer to the object is no longer required, you need to decrement the reference counter for the object by calling the `KosPutObject()` function from the objects.h API.

Unbinding an object from an identifier in the table

To unbind an object from an identifier in the table, call the `KosCloseObjHandle()` function. Decrements the reference counter for an object to balance the incrementation performed when calling the `KosCreateObjHandle()` function.

Traversing objects bound to identifiers in the table

To traverse objects bound to identifiers in the table, call the `KosWalkObjHandleTable()` function. Use the `callback` parameter to define the callback function that is called for each object during traversal and receives pointers to the object and data that is passed to the `KosWalkObjHandleTable()` function via the `context` parameter. The callback function must not call other API functions. If the callback function returns a value with the `KOS_OBJECT_WALK_FINISH` flag, the `KosWalkObjHandleTable()` function stops traversal. If a callback function returns a value with the `KOS_OBJECT_WALK_REMOVE_OBJECT` flag, the `KosWalkObjHandleTable()` function decrements the reference counter for the corresponding object. (The flags are defined in the header file `sysroot-*-kos/include/kos/objects.h` from the KasperskyOS SDK.)

Deleting the object identifier table

To delete the object identifier table, call the `KosDestroyObjHandleTable()` function. Prior to deleting the table, the function performs operations with the objects bound to identifiers in the table. If the `callback` parameter is set to `RTL_NULL`, the `KosDestroyObjHandleTable()` function decrements the reference counter for each object. If a callback function is defined via the `callback` parameter, the `KosDestroyObjHandleTable()` function calls this callback function for each object during traversal identically to the `KosWalkObjHandleTable()` function.

Information about API functions

objhandles.h functions

Function	Information about the function
`KosCreateObjHandleTable()`	Purpose Creates an object identifier table. Parameters • [in] `initialCount` – initial capacity of the table (number of identifiers). • [in] `maxCount` – maximum capacity of the table (number of identifiers). • [out] `createdTable` – pointer to the table address. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If the `createdTable` parameter is set to `RTL_NULL`, or the `maxCount` parameter is set to `0`, or the value in the `initialCount` parameter exceeds the value in the `maxCount` parameter, or the value in the `maxCount` parameter exceeds the maximum value, the function returns `rcInvalidArgument`. If there is insufficient memory to perform the required operations, the function returns `rcOutOfMemory`.
`KosCreateObjHandle()`	Purpose Binds an object to an identifier in the table. Parameters • [in] `table` – pointer to the object identifier table. • [in] `object` – pointer to the object. • [out] `createdHandle` – pointer to the object ID. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If any parameter is set to `RTL_NULL`, the function returns `rcInvalidArgument`. If there is insufficient memory to perform the required operations, the function returns `rcOutOfMemory`.
`KosRefObjectByObjHandle()`	Purpose Gets a pointer to an object based on an identifier in the table. Parameters • [in] `table` – pointer to the object identifier table. • [in] `handle` – object identifier. • [out] `refObject` – pointer to the object address. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If the `table` or `refObject` parameter is set to `RTL_NULL`, the function returns `rcInvalidArgument`. If the `handle` parameter contains an invalid object identifier, the function returns `rcResourceNotFound`.

Function	Information about the function
`KosCloseObjHandle()`	Purpose Unbinds an object from an identifier in the table. Parameters • [in] `table` – pointer to the object identifier table. • [in] `handle` – object identifier. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If the `table` parameter is set to `RTL_NULL`, the function returns `rcInvalidArgument`. If the `handle` parameter contains an invalid object identifier, the function returns `rcResourceNotFound`.
`KosWalkObjHandleTable()`	Purpose Traverses objects bound to identifiers in the table, and calls the defined function for each object during traversal. Parameters • [in] `table` – pointer to the object identifier table. • [in] `callback` – ID of the callback function that is called for each object during traversal. The parameter type is defined in the header file `sysroot-*-kos/include/kos/objects.h` from the KasperskyOS SDK. • [in,optional] `context` – pointer to the data that will be passed to the callback function that was defined via the `callback` parameter. You can specify `RTL_NULL` if data does not need to be passed. The callback function will receive the passed data through its own `context` parameter. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If the `table` or `callback` parameter is set to `RTL_NULL`, the function returns `rcInvalidArgument`.
`KosDestroyObjHandleTable()`	Purpose Deletes the object identifier table, and also (optionally) traverses objects bound to identifiers in the table and calls the defined function for each object during traversal. Parameters • [in] `table` – pointer to the object identifier table. • [in,optional] `callback` – ID of the callback function that is called for each object during traversal, or `RTL_NULL` if a callback function is not needed. The parameter type is defined in the header file `sysroot-*-kos/include/kos/objects.h` from the KasperskyOS SDK. • [in,optional] `context` – pointer to the data that will be passed to the callback function that was defined via the `callback` parameter. You can specify `RTL_NULL` if the `callback` parameter is set to `RTL_NULL` or if you do not need to pass data to a function defined via the `callback` parameter. The callback function will receive the passed data through its own `context` parameter. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If the `table` parameter is set to `RTL_NULL`, the function returns `rcInvalidArgument`. If the `callback` parameter is set to `RTL_NULL` but the `context` parameter is set to a value other than `RTL_NULL`, the function returns `rcInvalidArgument`.

Using KosStrings (strings.h)

The API is defined in the header file `sysroot-*-kos/include/kos/strings.h` from the KasperskyOS SDK.

The API lets you use `KosStrings`, which have the following distinguishing characteristics:

1. A `KosString` is a C-string (with a terminating zero) that follows the header containing the service data.

2. `KosStrings` are stored in hash tables, which are arrays whose elements are lists of strings. (The pointers for creating linked lists of strings are stored in the headers of strings.) Each list consists of one or more strings. Each list contains the strings who have the same value for the remainder after dividing the hash value of the string by the number of lists in the hash table. The main property of a hash table is that the computational complexity of searching for a string does not depend on the degree of occupancy of this table if all lists include only one string.

   When adding `KosStrings` to a hash table in real-world scenarios, lists containing multiple strings may be created. This leads to varying computational complexity when searching for strings in the hash table for the following reason. When searching for a key, the hash value is calculated and then converted into a string list index (by calculating the remainder from dividing the hash value by the number of lists). If there is only one string in the list containing the calculated index, the search will be complete after comparing this string to the key. However, if there are multiple strings in the list containing the calculated index, you may need to compare the key to all strings in this list to complete the search. In this case, the computational complexity of searching for a string may be higher than when there is only one string in the list.

3. `KosStrings` from one hash table are unique, which means that there are no identical strings in a hash table. If the strings from one hash table have different IDs, that means that these are different strings. (In the API, the ID of the `KosString` is the pointer to this string.) This lets you compare strings based on their IDs instead of their contents, thereby ensuring that the computational complexity of the comparison does not depend on the sizes of strings.

4. The size of a `KosString` is stored in the header instead of being calculated based on a search for a terminating zero, thereby ensuring that the computational complexity of getting the size does not depend on the actual size of the string.

5. `KosStrings` cannot be modified. This characteristic is a result of characteristics 2–4. Modification of a string may lead to the appearance of identical strings in the same hash table, a discrepancy between the actual size of a string and the size stored in the header, and a discrepancy between the string contents and the index of the string list in which this string is listed in the hash table.

6. A `KosString` exists as long as the reference counter for it remains greater than zero. (The string links counter is stored in the header.)

Information about API functions is provided in the table below.

Creating a hash table

You can use the default hash table, which is created automatically during initialization of the `libkos` library. This table may include no more than 2039 lists of strings with a string size of no more than 65,524 bytes without taking into account the terminating zero.

You can create a hash table with the required maximum number of string lists and the required maximum size of strings by calling the `KosCreateStringRoot()` function.

A hash table is created empty without any strings.

Adding a KosString to a hash table

To add a string to the default hash table, call the `KosCreateString()` function.

To add a string to a defined hash table, call the `KosCreateStringEx()` function.

If the string being added is already listed in the hash table, these functions do not add a new string but instead pass the ID of the already existing string via the `outStr` parameter.

Searching for a KosString in a hash table

To search for a string in the default hash table, call the `KosGetString()` function.

To search for a string in a defined hash table, call the `KosGetStringEx()` function.

The search is completed successfully if the key completely matches the string.

Managing the lifetime of KosStrings

A string exists as long as the reference counter for it remains greater than zero. The `KosCreateString()` and `KosCreateStringEx()` functions add a string whose links counter has a value of 1 to the hash table. If the string has already been added to the hash table, these functions increment the reference counter for this string. The `KosGetString()` and `KosGetStringEx()` functions increment the reference counter for the found string. This counter can then be incremented or decremented to manage the lifetime of the string. When passing the ID of a string to another program component, you must increment the reference counter for the corresponding string to ensure that this string will exist for the amount of time required by this component. If the string is no longer needed, you must decrement the reference counter for this string to ensure that it is destroyed when no other links remain.

To increment the reference counter for a string, call the `KosRefString()` or `KosRefStringEx()` function.

To decrement the reference counter for a string in the default hash table, call the `KosPutString()` function.

To decrement the reference counter for a string in a defined hash table, call the `KosPutStringEx()` function.

Getting the size of a KosString

To get the size of a string without taking into account the terminating zero, call the `KosGetStringLen()` function.

To get the size of a string while taking into account the terminating zero, call the `KosGetStringSize()` function.

Deleting a hash table

To delete a hash table, call the `KosDestroyStringRoot()` function. The table will be deleted only if it is empty.

Information about API functions

strings.h functions

Function	Information about the function
`KosCreateStringRoot()`	Purpose Creates an empty hash table. Parameters • [in] `numBuckets` – maximum number of `KosString` lists. • [in] `maxStrLength` – maximum size of `KosStrings` without taking into account the terminating zero (in bytes). Returned values: ID of the hash table.
`KosDestroyStringRoot()`	Purpose Deletes an empty hash table. Parameters • [in] `handle` – ID of the hash table. Returned values N/A
`KosGetString()`	Purpose Searches for a `KosString` in the default hash table. Parameters • [in] `str` – pointer to the C-string containing the search key. Returned values: If successful, the function returns the ID of the `KosString`, otherwise it returns `RTL_NULL`.
`KosGetStringEx()`	Purpose Searches for a `KosString` in a defined hash table. Parameters • [in,optional] `root` – ID of the hash table, or `RTL_NULL` to assign the default hash table. • [in] `str` – pointer to the C-string containing the search key. Returned values: If successful, the function returns the ID of the `KosString`, otherwise it returns `RTL_NULL`.

Function	Information about the function
`KosCreateString()`	Purpose Adds a `KosString` to the default hash table. Parameters • [in] `str` – pointer to the C-string whose data needs to be put into the `KosString`. • [out] `outStr` – pointer to the ID of the `KosString`. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosCreateStringEx()`	Purpose Adds a `KosString` to a defined hash table. Parameters • [in,optional] `root` – ID of the hash table, or `RTL_NULL` to assign the default hash table. • [in] `str` – pointer to the C-string whose data needs to be put into the `KosString`. • [out] `outStr` – pointer to the ID of the `KosString`. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosRefString()`	Purpose Increments the reference counter for a `KosString`. Parameters • [in] `str` – ID of the `KosString`. Returned values N/A
`KosRefStringEx()`	Purpose Increments the reference counter for a `KosString`. Parameters • [in,optional] `root` – fictitious parameter. You must specify `RTL_NULL`. • [in] `str` – ID of the `KosString`. Returned values N/A
`KosPutString()`	Purpose Decrements the reference counter for a `KosString` in the default hash table. Parameters • [in] `str` – ID of the `KosString`. Returned values N/A

Function	Information about the function
`KosPutStringEx()`	Purpose Decrements the reference counter for a `KosString` in a defined hash table. Parameters • [in,optional] `root` – ID of the hash table, or `RTL_NULL` to assign the default hash table. • [in] `str` – ID of the `KosString`. Returned values N/A
`KosGetStringLen()`	Purpose Gets the size of the `KosString` without taking into account the terminating zero. Parameters • [in] `str` – ID of the `KosString`. Returned values: Size of the `KosString` without taking into account the terminating zero (in bytes). Additional information If the parameter has the `RTL_NULL` value, it returns `0`.
`KosGetStringSize()`	Purpose Gets the size of the `KosString` while taking into account the terminating zero. Parameters • [in] `str` – ID of the `KosString`. Returned values: Size of the `KosString` while taking into account the terminating zero (in bytes). Additional information If the parameter has the `RTL_NULL` value, it returns `0`.

KasperskyOS kernel XHCI DbC driver management (xhcidbg_api.h)

This API is defined in the header file `sysroot-*-kos/include/coresrv/xhcidbg/xhcidbg_api.h` from the KasperskyOS SDK.

This API is intended for use in the code of drivers that implement a USB stack (kusb class drivers) and lets you temporarily suspend the XHCI DbC (Debug Capability) driver of the KasperskyOS kernel. (The KasperskyOS kernel includes the XHCI DbC driver, which is a simplified XHCI controller driver and is used by the kernel for diagnostic output or by the kernel GDB server to interact with the GDB debugger.) If a kusb class driver needs to restart the XHCI controller, the kernel's XHCI DbC driver must be suspended by calling the `KnXhcidbgSuspend()` function before the restart and then started by calling the `KnXhcidbgResume()` function after the restart. If you do not do this, a restart of the XHCI controller will lead to a failure of diagnostic output (or debugging) via the USB port.

> This API has not been implemented for the majority of hardware platforms that are supported by KasperskyOS (the functions return `rcUnimplemented`).

Information about API functions is provided in the table below.

xhcidbg_api.h functions

Function	Information about the function
`KnXhcidbgSuspend()`	Purpose Suspends the XHCI DbC driver of the KasperskyOS kernel. Parameters N/A Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnXhcidbgResume()`	Purpose Starts a suspended XHCI DbC driver of the KasperskyOS kernel. Parameters N/A Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Getting IPC handles and RIIDs to use statically created IPC channels (sl-static.h)

This API is defined in the header file `sysroot-*-kos/include/coresrv/sl/sl-static.h` from the KasperskyOS SDK.

The API allows servers to get the listener handles and allows clients to get the client IPC handles and endpoint IDs (RIIDs) for the purpose of using statically created IPC channels. (For information about using client and server IPC handles and endpoint IDs (RIIDs), see Initializing IPC transport for interprocess communication and managing IPC request processing (transport-kos.h, transport-kos-dispatch.h).)

> Closing an IPC handle received via the API will cause the IPC channel to become unavailable. After an IPC handle is closed, it is impossible to obtain it again or restore access to the IPC channel.

Information about API functions is provided in the tables below.

Using the API on the server side

To get the listener handle based on the name of a statically created IPC channel connected to a server, call the `SLStaticResolveHandle()` function.

To get the names and listener handles for all statically created IPC channels connected to a server, call the `SLStaticGetServerChannels()` or `SLStaticAllocateAndGetServerChannels()` function. The second function creates a buffer for output data. To flush this buffer, call the `SLStaticFreeServerChannels()` function.

Using the API on the client side

To get a client IPC handle based on the name of a statically created IPC channel connected to the client, call the `SLStaticResolveHandle()` function.

To get an endpoint ID (RIID) based on the qualified name of the endpoint provided via the statically created IPC channel connected to the client, call the `SLStaticResolveRiid()` function. (For information about the qualified name of an endpoint, see Binding methods of security models to security events.)

To get information about the endpoints provided by servers that are connected to a client via statically created IPC channels, call the `SLStaticGetEndpoints()` or `SLStaticAllocateAndGetEndpoints()` function. This information includes the names of servers, the names of endpoints (from the EDL descriptions and CDL descriptions, and qualified names) and endpoint IDs (RIIDs). You can define the following criteria for filtering endpoints:

- Server name.
- Qualified name of the component instance.

  For information about the qualified name of a component instance, see Binding methods of security models to security events.

  The qualified name of a component instance lets you explicitly identify a component instance implemented on a server. To get information about component instances that are implemented on servers that are connected to a client via statically created IPC channels, call the `SLStaticGetComponents()` function. This information includes the names of servers and the names of component instances (from the EDL descriptions and CDL descriptions, and qualified names).
- Interface name of endpoints according to the IDL description.
- Boolean value that defines whether you need to receive information about endpoints provided by embedded instances of components.

The `SLStaticAllocateAndGetEndpoints()` function creates a buffer for output data. To flush this buffer, call the `SLStaticFreeEndpoints()` function.

Getting a process name

The API additionally includes the `SLStaticSelfUniqId()` function, which lets you get the name of the calling process. This function is intended for use by the `libkos` library.

Information about API functions

sl-static.h functions

Function	Information about the function
`SLStaticResolveHandle()`	Purpose Gets an IPC handle based on the name of a statically created IPC channel connected to a calling client or server. Parameters • [in] `channelName` – pointer to the name of the statically created IPC channel. • [in] `type` – value that defines which IPC handle should be received: client (on the client side) or listener (on the server side). Returned values: If successful, it returns the IPC handle. Otherwise, it returns `INVALID_HANDLE`.
`SLStaticResolveRiid()`	Purpose Gets an endpoint ID (RIID) based on the qualified name of an endpoint provided via a statically created IPC channel connected to a calling client. Parameters • [in] `handle` – client IPC handle of the statically created IPC channel. • [in] `name` – pointer to the qualified name of the endpoint. Returned values: If successful, the function returns the endpoint ID (RIID), otherwise it returns `INVALID_RIID`. The type of returned value is defined in the header file `sysroot-*-kos/include/nk/types.h` from the KasperskyOS SDK.
`SLStaticSelfUniqId()`	Purpose Gets the name of a calling process. Parameters N/A Returned values If successful, the function returns the pointer to the name of the calling process, otherwise it returns `RTL_NULL`.

Function	Information about the function
`SLStaticGetServerChannels()`	Purpose Gets the names and listener handles for all statically created IPC channels connected to a calling server. Parameters • [in] `maxChannelsCount` – maximum number of elements in the array defined via the `channels` parameter. • [out] `channels` – pointer to the array of structures used to save the names and listener handles for statically created IPC channels. • [out] `channelsCount` – pointer to the actual number of statically created IPC channels connected to the calling server. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If the value received via the `channelsCount` parameter exceeds the value defined via the `maxChannelsCount` parameter, the function returns `rcBufferTooSmall`.
`SLStaticAllocateAndGetServerChannels()`	Purpose Creates an array and uses this array to store the names and listener handles for all statically created IPC channels connected to a calling server. Parameters • [out] `outChannels` – pointer to the address of the array of structures containing the names and listener handles for statically created IPC channels. • [out] `channelsCount` – pointer to the number of statically created IPC channels connected to the calling server. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`SLStaticFreeServerChannels()`	Purpose Deletes an array that was created by the `SLStaticAllocateAndGetServerChannels()` function call. Parameters • [in] `channels` – pointer to the array that was created by the `SLStaticAllocateAndGetServerChannels()` function call. Returned values N/A

Function	Information about the function
`SLStaticGetComponents()`	Purpose Gets information about component instances that are implemented on servers that are connected to a calling client via statically created IPC channels. Parameters • [in] `componentType` – pointer to the component name according to the CDL description. • [in] `maxCount` – number of elements in the array defined via the `components` parameter. • [out] `components` – pointer to the array of structures used to save the names of servers and the names of component instances (from the EDL descriptions and CDL descriptions, and qualified names). • [out] `componentsCount` – pointer to the actual number of component instances that are implemented on servers connected to a calling client via statically created IPC channels. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If the value received via the `componentsCount` parameter exceeds the value defined via the `maxCount` parameter, the function returns `rcBufferTooSmall`.
`SLStaticGetEndpoints()`	Purpose Gets information about endpoints provided by servers that are connected to a calling client via statically created IPC channels. Parameters • [in,optional] `serverName` – pointer to the server name, or `RTL_NULL` if you do not need to define a server with a specific name. • [in,optional] `componentPath` – pointer to the qualified name of the component instance, or `RTL_NULL` if you do not need to define a specific component instance. • [in,optional] `endpointType` – pointer to the interface name of endpoints according to the IDL description, or `RTL_NULL` if you do not need to define a specific interface of endpoints. • [in] `recursive` – Boolean value that defines whether or not you need to receive information about endpoints provided by embedded instances of components (`rtl_true` – yes, `rtl_false` – no). • [in] `maxCount` – number of elements in the array defined via the `endpoints` parameter. • [out] `endpoints` – pointer to the array of structures used to save the names of servers, the names of endpoints (from the EDL descriptions and CDL descriptions, and qualified names) and endpoint IDs (RIIDs). • [out] `endpointsCount` – pointer to the actual number of endpoints that are provided by servers connected to a calling client via statically created IPC channels and that match the filtering criteria defined via the `serverName`, `componentPath`, `endpointType` and `recursive` parameters. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If the value received via the `endpointsCount` parameter exceeds the value defined via the `maxCount` parameter, the function returns `rcBufferTooSmall`.

Function	Information about the function
`SLStaticAllocateAndGetEndpoints()`	Purpose Creates an array and uses this array to store information about endpoints provided by servers that are connected to a calling client via statically created IPC channels. Parameters • [in,optional] `serverName` – pointer to the server name, or `RTL_NULL` if you do not need to define a server with a specific name. • [in,optional] `componentPath` – pointer to the qualified name of the component instance, or `RTL_NULL` if you do not need to define a specific component instance. • [in,optional] `endpointType` – pointer to the interface name of endpoints according to the IDL description, or `RTL_NULL` if you do not need to define a specific interface of endpoints. • [in] `recursive` – Boolean value that defines whether or not you need to receive information about endpoints provided by embedded instances of components (`rtl_true` – yes, `rtl_false` – no). • [out] `outEndpoints` – pointer to the address of the array of structures containing the names of servers, the names of endpoints (from the EDL descriptions and CDL descriptions, and qualified names) and endpoint IDs (RIIDs). • [out] `outEndpointsCount` – pointer to the number of endpoints that are provided by servers connected to a calling client via statically created IPC channels and that match the filtering criteria defined via the `serverName`, `componentPath`, `endpointType` and `recursive` parameters. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`SLStaticFreeEndpoints()`	Purpose Deletes an array that was created by a `SLStaticAllocateAndGetEndpoints()` function call. Parameters • [in] `endpoints` – pointer to the array that was created by a `SLStaticAllocateAndGetEndpoints()` function call. Returned values N/A

Power management (pm_api.h)

This API is defined in the header file `sysroot-*-kos/include/coresrv/pm/pm_api.h` from the KasperskyOS SDK.

The API changes the power mode of the hardware platform (for example, shutdown, restart, or hibernate), and enables and disables processors (computing cores).

Information about API functions is provided in the table below.

Changing the power mode of the hardware platform

To make a request to change the power mode of the hardware platform, call the `KnPmRequest()` function.

Enabling and disabling processors (computing cores)

The `KnPmSetCpusOnline()` and `KnPmGetCpusOnline()` functions have the `mask` parameter, which is used to pass the CPU mask. A *CPU mask* is a bit mask indicating the set of processors (processor cores). A flag set in the *i*th bit indicates that a processor with the *i* index is included in the set (numbering starts at zero). To work with a CPU mask, you must use the functions defined in the header file `sysroot-*-kos/include/rtl/cpuset.h`.

To enable and/or disable processors, call the `KnPmSetCpusOnline()` function while specifying the CPU mask indicating the required set of active processors. The flag corresponding to the bootstrap processor must be set in this mask, otherwise the function will end with an error. To get the bootstrap processor index, call the `KnPmGetBootstrapCpuNum()` function.

To find out which processors are in the active state, call the `KnPmGetCpusOnline()` function. The output parameter of this function is the CPU mask indicating the actual set of active processors.

If the `KnPmSetCpusOnline()` function ended with an error, this could lead to an unpredictable change in the set of active processors. To make a repeated attempt to configure the set of active processors, you must find out which processors are in the active state by calling the `KnPmGetCpusOnline()` function and use this information to adjust the required set of active processors during the subsequent `KnPmSetCpusOnline()` function call.

Information about API functions

pm_api.h functions

Function	Information about the function
`KnPmRequest()`	Purpose Requests to change the power mode of a hardware platform. Parameters • [in] `request` – value defining the necessary power mode of the hardware platform. The parameter type is defined in the header file `sysroot-*-kos/include/pm/pmstate.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnPmSetCpusOnline()`	Purpose Requests to enable and/or disable processors. Parameters • [in] `mask` – pointer to the CPU mask indicating the required set of active processors. The type of CPU mask is defined in the header file `sysroot-*-kos/include/rtl/cpuset.h` from the KasperskyOS SDK. • [in] `timeoutMs` – fictitious parameter (obsolete). Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KnPmGetCpusOnline()`	Purpose Gets information regarding which processors are in the active state. Parameters • [out] `mask` – pointer to the CPU mask indicating the actual set of active processors. The type of CPU mask is defined in the header file `sysroot-*-kos/include/rtl/cpuset.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnPmGetBootstrapCpuNum()`	Purpose Gets the index of the bootstrap processor. Parameters • [out] `num` – pointer to the bootstrap processor index. The type of processor index is defined in the header file `sysroot-*-kos/include/rtl/cpuset.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Getting values of performance counters (value_api.h)

API is defined in the header file `sysroot-*-kos/include/coresrv/stat/value_api.h` from the KasperskyOS SDK.

This API lets you receive the values of performance counters, such as counters for the uptime, sizes of memory and objects.

Information about API functions is provided in the table below.

Using the API

To receive the values of performance counters, use the following functions:

- `KnStatGetValueAsInt64()` – get the value of the performance counter in 64-bit signed integer format.
- `KnStatGetValueAsUInt64()` – get the value of the performance counter in 64-bit unsigned integer format.
- `KnStatGetValues()` – get the set of values of performance counters.
- `KnStatGetObjectValues()` – get the set of values of performance counters associated with a system resource (process or thread).

The `KnStatGetValueAsInt64()` and `KnStatGetValueAsUInt64()` functions use the `name` parameter to receive the name of the performance counter. The `KnStatGetValues()` and `KnStatGetObjectValues()` functions use the `names` parameter to receive an array containing the names of the performance counters. A performance counter name is a string consisting of one or more segments separated by a dot (see the tables containing the names and values of performance counters below).

The optional `prefix` parameter is used by the `KnStatGetValueAsInt64()` and `KnStatGetValueAsUInt64()` functions to receive the prefix for the performance counter name defined via the `name` parameter, and the `KnStatGetValues()` function receives the prefix for the performance counter names defined via the `names` parameter. This prefix is the initial part of the performance counter name consisting of one or more segments separated by a dot, and it ends with a dot. The `prefix` parameter is convenient to use when you need to receive the values of multiple performance counters with names that have an identical initial or final part. The prefix also may contain an identical or different initial part.

You can use the `KnStatGetObjectValues()` function to receive the values of only those performance counters that are associated with a process or thread defined via the `handle` parameter. However, the names of performance counters associated with a process must be specified without the initial part `task.<PID>.`, and the names of performance counters associated with a thread must be specified without the initial part `task.<PID>.thread.<TID>.`

Information about API functions

value_api.h functions

Function	Information about the function
`KnStatGetValueAsInt64()`	Purpose Gets the value of a performance counter in 64-bit signed integer format. Parameters • [in,optional] `prefix` – pointer to the prefix for the performance counter name, or `RTL_NULL` if you do not need to use a prefix. • [in] `name` – pointer to the name of the performance counter. • [out] `value` – pointer to the value of the performance counter. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnStatGetValueAsUInt64()`	Purpose Gets the value of a performance counter in 64-bit unsigned integer format. Parameters • [in,optional] `prefix` – pointer to the prefix for the performance counter name, or `RTL_NULL` if you do not need to use a prefix. • [in] `name` – pointer to the name of the performance counter. • [out] `value` – pointer to the value of the performance counter. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KnStatGetValues()`	Purpose Gets the values of performance counters. Parameters • [in,optional] `prefix` – pointer to the prefix for the performance counter names, or `RTL_NULL` if you do not need to use a prefix. • [in] `names` – pointer to the array containing the names of performance counters. • [in] `count` – quantity of performance counters. • [out] `values` – pointer to the array of structures containing the values of performance counters. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnStatGetObjectValues()`	Purpose Gets the values of performance counters associated with a system resource (process or thread). Parameters • [in] `handle` – handle of the system resource. • [in] `names` – pointer to the array containing the names of performance counters. • [in] `count` – quantity of performance counters. • [out] `values` – pointer to the array of structures containing the values of performance counters. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Names and values of performance counters

Uptime counters have values in nanoseconds. Uptime is counted from the startup of the KasperskyOS kernel. Most memory size counters have values defined in memory pages. The size of a memory page is 4 KB for all hardware platforms supported by KasperskyOS. The names of performance counters associated with a process contain the process identifier (PID). The names of performance counters associated with a thread contain the process identifier (PID) and thread identifier (TID). The qualified names of core endpoints can be viewed in Methods of KasperskyOS core endpoints.

System performance counters

Counter name	Counter value
`tasks`	Number of processes (excluding the kernel process)
`threads`	Number of threads (including kernel threads)
`time.user`	Total uptime of all processors in user mode
`time.kernel`	Total uptime of all processors in kernel mode
`time.idle`	Total uptime of all processors in idle mode
`time.sleep.ipc`	Total locking time of all threads awaiting IPC
`time.sleep.futex`	Total locking time of all threads resulting from futex use
`time.sleep.voluntary`	Total locking time of all threads resulting from calls of the `KosThreadSleep()` and `KnSleep()` functions

Counter name	Counter value
`time.sleep.other`	Total locking time of all threads due to reasons not related to pending IPC, futex use, or calls of the `KosThreadSleep()` and `KnSleep()` functions
`mem.total`	Size of installed physical memory, in memory pages
`mem.free`	Size of free physical memory, in memory pages
`mem.shmem`	Size of physical memory used as shared memory, in memory pages
`mem.exsize`	Size of physical memory with the "execution access" attribute, in memory pages
`mem.kmalloc.total`	Size of physical memory reserved for kernel allocator distribution, in memory pages
`mem.kmalloc.free`	Size of free physical memory reserved for kernel allocator distribution, in memory pages
`mem.kmalloc.count`	Quantity of physical memory regions distributed by the kernel allocator
`mem.kmalloc.size`	Size of physical memory distributed by the kernel allocator (in bytes)
`mem.kmalloc.cached`	Size of free physical memory that is reserved for kernel allocator distribution and is not available on every processor (computing core), but instead is available only separately on each processor, in bytes
`dma.count`	Quantity of DMA buffers
`dma.allocated`	Total size of all DMA buffers, in memory pages
`romfs.is_initial`	Indicator of ROMFS image in kernel memory (`rtl_true` or `1` – yes, `rtl_false` or `0` – no)
`romfs.files.count`	Quantity of files in ROMFS image
`romfs.size`	Size of ROMFS image, in memory pages
`objects.total`	Quantity of kernel objects
`handles.total`	Quantity of handles
`handles.tables.size`	Total size of handle tables, including used and unused identifiers (in bytes)
`sid.used`	Quantity of security identifiers (SIDs)
`sid.capacity`	Maximum possible quantity of security identifiers (SIDs)
`ipc.recv`	Quantity of `Recv()` system calls executed by all threads
`ipc.reply`	Quantity of `Reply()` system calls executed by all threads
`ipc.call`	Quantity of `Call()` system calls executed by all threads

Performance counters associated with a processor (computing core)

Counter name	Counter value
`cpu.<processor index>.time.user`	CPU uptime in user mode
`cpu.<processor index>.time.kernel`	CPU uptime in kernel mode
`cpu.<processor index>.time.idle`	CPU uptime in idle mode
`cpu.<processor index>.conswitches`	Quantity of thread context switches on the processor
`cpu.<processor index>.irqs`	Quantity of interrupts handled on the processor
`cpu.<processor index>.irq.clock`	Quantity of timer interrupts handled on the processor
`cpu.<processor index>.irq.dsptch`	Quantity of thread rescheduling interrupts handled on the processor

Performance counters associated with interrupts

Counter name	Counter value
`irqs.count`	Maximum number of hardware interrupts that can be registered, including MSI interrupts and interrupts that occur when devices query the interrupt controller via hardware interrupt lines
`irq.<interrupt number>.cpu.<processor index>`	Quantity of interrupts received by the processor (computing core)
`irq.<interrupt number>.ipl`	Priority of the interrupt

Counter name	Counter value
`irq.<interrupt number>.enabled`	Indicator that at least one thread is attached to the interrupt (`rtl_true` or `1` – yes, `rtl_false` or `0` – no)
`irq.<interrupt number>.handlers`	Quantity of interrupt handlers
`irq.<interrupt number>.mode`	Interrupt conditions (according to the `irqmode_t` list defined in the header file `sysroot-*-kos/include/hal/irqmode.h` from the KasperskyOS SDK)

Performance counters associated with a process

Counter name	Counter value
`task.<PID>.state`	State of the process (according to the list of `TaskExecutionStates` defined in the header file `sysroot-*-kos/include/task/pcbpage.h` from the KasperskyOS SDK)
`task.<PID>.threads`	Number of threads in the process
`task.<PID>.start`	Time between kernel startup and loading of the program image into process memory
`task.<PID>.imsize`	Size of program image in process memory, in bytes
`task.<PID>.ppid`	Parent process ID (PPID)
`task.<PID>.prio`	Priority of the initial thread of the process
`task.<PID>.allocated`	Size of physical memory used by the process, in memory pages (describes only the memory that is allocated directly for the process; for example, if the memory of a process is mapped to an MDL buffer created by another process, the size of this buffer is not included in this value)
`task.<PID>.vmsize`	Size of process virtual memory, in memory pages
`task.<PID>.shmem`	Size of process virtual memory mapped to shared physical memory, in memory pages
`task.<PID>.ptsize`	Size of process pages table, in memory pages
`task.<PID>.dma.mapped`	Size of DMA memory mapped to process virtual memory, in memory pages
`task.<PID>.mmio.mapped`	Size of MMIO memory mapped to process virtual memory, in memory pages
`task.<PID>.time.user`	Total execution time of all process threads in user mode
`task.<PID>.time.kernel`	Total execution time of all process threads in kernel mode
`task.<PID>.time.sleep.ipc`	Total locking time of all process threads awaiting IPC
`task.<PID>.time.sleep.futex`	Total locking time of all process threads resulting from futex use
`task.<PID>.time.sleep.voluntary`	Total locking time of all process threads resulting from calls of the `KosThreadSleep()` and `KnSleep()` functions
`task.<PID>.time.sleep.other`	Total locking time of all process threads due to reasons not related to pending IPC, futex use, or calls of the `KosThreadSleep()` and `KnSleep()` functions
`task.<PID>.handles.allocated`	Quantity of handles owned by the process
`task.<PID>.handles.table.size`	Size of process handles table, including used and unused identifiers (in bytes)
`task.<PID>.handles.table.capacity`	Quantity of handles that can be accommodated in the process handles table
`task.<PID>.objects.user`	Quantity of user resource handles created by the process
`task.<PID>.ipc.recv`	Quantity of `Recv()` system calls executed by all process threads
`task.<PID>.ipc.reply`	Quantity of `Reply()` system calls executed by all process threads
`task.<PID>.ipc.call`	Quantity of `Call()` system calls executed by all process threads

Performance counters associated with a thread

Counter name	Counter value
`task.<PID>.thread.<TID>.time.user`	Execution time of process thread in user mode
`task.<PID>.thread.<TID>.time.kernel`	Execution time of process thread in kernel mode

Counter name	Counter value
`task.<PID>.thread.<TID>.time.sleep.ipc`	Locking time of process thread awaiting IPC
`task.<PID>.thread.<TID>.time.sleep.futex`	Locking time of process thread resulting from futex use
`task.<PID>.thread.<TID>.time.sleep.voluntary`	Locking time of process thread resulting from calls of the `KosThreadSleep()` and `KnSleep()` functions
`task.<PID>.thread.<TID>.time.sleep.other`	Locking time of process thread due to reasons not related to pending IPC, futex use, or calls of the `KosThreadSleep()` and `KnSleep()` functions
`task.<PID>.thread.<TID>.conswitches`	Quantity of process thread evictions by other threads
`task.<PID>.thread.<TID>.sched.prio`	Priority of the thread.
`task.<PID>.thread.<TID>.sched.policy`	Thread scheduler class (in accordance with the `ThreadSchedPolicy` enumeration defined in the header file `sysroot-*-kos/include/thread/tidtype.h` from the KasperskyOS SDK)
`task.<PID>.thread.<TID>.ipc.recv`	Quantity of `Recv()` system calls executed by process thread
`task.<PID>.thread.<TID>.ipc.reply`	Quantity of `Reply()` system calls executed by process thread
`task.<PID>.thread.<TID>.ipc.call`	Quantity of `Call()` system calls executed by process thread

Using the hardware abstraction layer (hal_api.h, hal_common.h, hal_env.h, hal_privregs.h)

APIs are defined in the header files `sysroot-*-kos/include/coresrv/hal/_*.h` from the KasperskyOS SDK. The `hal_api.h` header file contains directives for including the rest of the `hal_*.h` header files.

Main capabilities of the API:

- Flush the processor cache.
- Get arrays of hardware-generated random byte values.
- Restart the hardware platform.
- Forward diagnostic data to the logging system.
- Get the values of hardware platform parameters.
- Get the number of processors (computing cores) on the hardware platform.
- Get and set the values of privileged registers of peripheral devices.

Information about API functions is provided in the tables below.

Flushing the processor cache

To clear the processor cache, call the `KnHalFlushCache()` function that is declared in the `hal_api.h` header file. This function flushes the processor cache of all accessible levels. You can also clear the data cache and instructions cache corresponding to the defined virtual memory region. For example, a driver must clear the processor data cache corresponding to the memory used for DMA because the caching process may cause the contents of this memory from the driver's perspective to differ from the actual contents visible to the device.

You can also flush the processor cache by using the `KosCpuCacheFlush()` function from the cpucache.h API. In contrast to the `KnHalFlushCache()` function, which always executes a system call to flush the processor cache, the `KosCpuCacheFlush()` function does this only if there is no capability to flush the processor cache without this system call.

Getting arrays of random byte values

To get an array of hardware-generated random byte values, call the `KnHalGetEntropy()` function that is declared in the `hal_api.h` header file.

To get information about the source of entropy (min-entropy value), call the `KnHalGetEntropyInfo()` function that is declared in the `hal_api.h` header file.

Restarting the hardware platform

To restart the hardware platform, call the `KnHalResetSystem()` function that is declared in the `hal_common.h` header file.

Forwarding diagnostic data to the logging system

To forward diagnostic data to the logging system, call the `KnHalDebugWrite()` function that is declared in the `hal_common.h` header file. (The logging system prints diagnostic data, for example, via UART or saves it to a file.)

Getting the values of hardware platform parameters

To get the value of a hardware platform parameter, call the function `KnHalGetEnv()` or `KnHalGetEnvMemRegion()` defined in the header file `hal_env.h`. The values of hardware platform parameters contain data, such as information about the motherboard, architecture, CPU frequency, SMBIOS (System Management BIOS), and FDT (Flattened Device Tree). Symbolic constants containing the names of parameters are defined in the header file `sysroot-*-kos/include/hal/env_names.h` from the KasperskyOS SDK. The `KnHalGetEnvMemRegion()` function can be used only to get information about the SMBIOS and FDT (the values of parameters named `HAL_ENV_SMBIOS` and `HAL_ENV_GET_FDT_INFO`, respectively).

Getting the number of processors (computing cores) on the hardware platform

To get the number of processors on the hardware platform, call the `KnHalGetCpuCount()` function that is declared in the `hal_env.h` header file.

Getting and setting the values of privileged registers of peripheral devices

To get or set the value of the privileged register of a peripheral device, use the functions declared in the `hal_privregs.h` header file.

To get the value of a register, call the `KnHalGetPrivReg()` or `KnHalGetPrivRegRange()` function.

To define a register value, call the `KnHalSetPrivReg()` or `KnHalSetPrivRegRange()` function.

Information about API functions

hal_api.h functions

Function	Information about the function
`KnHalFlushCache()`	Purpose Flushes the processor cache. Parameters • [in] `type` – value defining the cache type (data cache, instructions cache, or joint data and instructions cache). The parameter type is defined in the header file `sysroot-*-kos/include/rtl/cpucache.h` from the KasperskyOS SDK. • [in] `va` – pointer to the virtual memory region. The cache corresponding to this region will be flushed. • [in] `size` – size (in bytes) of the virtual memory region. The cache corresponding to this region will be flushed. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnHalGetEntropy()`	Purpose Gets an array of hardware-generated random byte values. Parameters • [out] `buffer` – pointer to the buffer for saving the array. • [in] `size` – size (in bytes) of the buffer used to store the array. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KnHalGetEntropyInfo()`	Purpose Gets information about the source of entropy. Parameters • [out] `info` – pointer to the structure containing the min-entropy value. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

hal_common.h functions

Function	Information about the function
`KnHalResetSystem()`	Purpose Restarts the hardware platform. Parameters N/A Returned values If the function has not been implemented, `rcUnimplemented` is returned.
`KnHalDebugWrite()`	Purpose Forwards diagnostic data to the logging system. Parameters • [in] `buf` – pointer to the diagnostic data. • [in] `size` – size (in bytes) of the diagnostic data. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

hal_env.h functions

Function	Information about the function
`KnHalGetEnv()`	Purpose Gets the value of a hardware platform parameter. Parameters • [in] `name` – pointer to the parameter name. • [out] `value` – pointer to the buffer used to save the parameter value. • [in] `size` – size of the buffer defined via the `value` parameter (in bytes). Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If the size of the buffer defined via the `value` parameter is insufficient, the function returns `rcBufferTooSmall`.

Function	Information about the function
`KnHalGetEnvMemRegion()`	Purpose Gets the value of a hardware platform parameter in the format `0x<base physical address of memory region> 0x<memory region size>\0` and gets the base physical address and size of the memory region from this value. Parameters • [in] `name` – pointer to the parameter name. • [out] `addr` – pointer to the base physical address of the memory region. • [out] `size` – pointer to the size of the memory region (in bytes). Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnHalGetCpuCount()`	Purpose Gets the number of processors (computing cores) on the hardware platform. Parameters • [out] `count` – pointer to the number of processors. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

hal_privregs.h functions

Function	Information about the function
`KnHalGetPrivReg()`	Purpose Gets the value of a peripheral device privileged register. Parameters • [in] `reg` – pointer to the register name. • [out] `val` – pointer to the register value. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnHalSetPrivReg()`	Purpose Sets the value of a peripheral device privileged register. Parameters • [in] `reg` – pointer to the register name. • [in] `val` – value of the register. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KnHalGetPrivRegRange()`	Purpose Gets the value of a peripheral device privileged register. Parameters • [in] `regRange` – pointer to the name of the registers range. • [in] `offset` – register offset in the registers range. • [out] `val` – pointer to the register value. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If the string defined via the `regRange` parameter has a length greater than the permitted length, the function returns `rcBufferTooLarge`. If an IPC error occurred, the function returns `rcFail`. If the name of the registers range is incorrect or if the register offset is outside of the permissible range, the function returns `rcInvalidArgument`. If the thread in which the function is called is attached to an interrupt, the function returns `rcInvalidThreadState`. (For details about attaching a thread to an interrupt, see Managing interrupt processing (irq.h).)
`KnHalSetPrivRegRange()`	Purpose Sets the value of a peripheral device privileged register. Parameters • [in] `regRange` – pointer to the name of the registers range. • [in] `offset` – register offset in the registers range. • [in] `val` – value of the register. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If the string defined via the `regRange` parameter has a length greater than the permitted length, the function returns `rcBufferTooLarge`. If an IPC error occurred, the function returns `rcFail`. If the name of the registers range is incorrect or if the register offset is outside of the permissible range, the function returns `rcInvalidArgument`. If the thread in which the function is called is attached to an interrupt, the function returns `rcInvalidThreadState`. (For details about attaching a thread to an interrupt, see Managing interrupt processing (irq.h).)

Flushing the processor cache and getting processor cache info (cpucache.h)

The API is defined in the header file `sysroot-*-kos/include/kos/cpucache.h` from the KasperskyOS SDK.

The API lets you flush the processor cache and get information about the processor cache.

Information about API functions is provided in the table below.

Flushing the processor cache

To flush the processor cache, call the `KosCpuCacheFlush()` function. This function flushes the processor cache of all accessible levels. You can also clear the data cache and instructions cache corresponding to the defined virtual memory region. For example, a driver must clear the processor data cache corresponding to the memory used for DMA because the caching process may cause the contents of this memory from the driver's perspective to differ from the actual contents visible to the device.

You can also flush the processor cache by using the `KnHalFlushCache()` function from the hal_api.h API. You must also keep in mind that the `KnHalFlushCache()` function always executes a system call to flush the processor cache, while the `KosCpuCacheFlush()` function does this only if there is no capability to flush the processor cache without this system call.

Getting processor cache info

To get information about the processor cache, call the `KosCpuCacheGetInfo()` function. This information includes the maximum and minimum sizes (bytes) of cache lines and information about the cache implementation restrictions on the utilized processor architecture. This information is needed when implementing low-level mechanisms for managing the processor cache, such as in drivers. For example, when implementing these mechanisms, you may need to determine the boundaries of cache lines or find out whether you can manage the cache in user mode.

A maximum and minimize size of cache lines could be required if the computing platform might include computing cores with different cache line sizes or an external processor cache controller.

Cache implementation restrictions are divided into general restrictions for all processor architectures and specific restrictions for different processor architectures. Flags that indicate general restrictions are defined in the header file `sysroot-*-kos/include/rtl/cpucache.h` from the KasperskyOS SDK. Flags that indicate specific restrictions are defined in the header files `sysroot-*-kos/include/rtl/<processor architecture>/cpucache.h` from the KasperskyOS SDK (for example, `sysroot-*-kos/include/rtl/arm64/cpucache.h`).

Information about API functions

cpucache.h functions

Function	Information about the function
`KosCpuCacheFlush()`	Purpose Flushes the processor cache. Parameters • [in] `type` – value defining the cache type (data cache, instructions cache, or joint data and instructions cache). The parameter type is defined in the header file `sysroot-*-kos/include/rtl/cpucache.h` from the KasperskyOS SDK. • [in] `va` – pointer to the virtual memory region. The cache corresponding to this region will be flushed. • [in] `size` – size (in bytes) of the virtual memory region. The cache corresponding to this region will be flushed. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosCpuCacheGetInfo()`	Purpose Gets information about the processor cache. Parameters • [out] `outInfo` – pointer to the structure containing information about the processor cache. The structure type is defined in the header file `sysroot-*-kos/include/rtl/cpucache.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Using the ROMFS file system (romfs.h)

The API is defined in the header file `sysroot-*-kos/include/kos/romfs.h` from the KasperskyOS SDK.

The API is designed to work with ROMFS file systems, which includes the ROMFS image supplied as part of the KasperskyOS kernel image and the ROMFS image placed in memory by the bootloader (similar to initrd in Linux). The mechanism that implements the API does not use VFS.

The input data and output data of some API functions are associated with blobs because the romfs.h API implementation uses the blob_api.h API, which is designed to gain access to named blobs.

Information about API functions is provided in the table below.

Mounting a ROMFS image

To mount a ROMFS image, call the `KosRomFsMount()` or `KosRomFsMountByHandle()` function.

You must use the `romFsName` parameter of the `KosRomFsMount()` or `KosRomFsMountByHandle()` function to specify an arbitrary mount point name whose size must not exceed the value of the `KOS_ROMFS_MAX_NAME` constant (not counting the terminating zero). In one process, each mount point must have a unique name.

You must use the `blobName` parameter of the `KosRomFsMount()` function to specify the name of the blob containing the required ROMFS image: `romfs.0` or `romfs.1`. In the first case, the ROMFS image will be mounted from the kernel image. In the second case, the ROMFS image that was placed in memory by the bootloader will be mounted.

In the `blob` parameter of the `KosRomFsMountByHandle()` function, you must specify the handle of the blob containing the required ROMFS image. This handle must be obtained by calling the `KnBlobGet()` function from the blob_api.h API. You must use the `name` parameter of the `KnBlobGet()` function to define the name of the blob containing the required ROMFS image: `romfs.0` or `romfs.1` (similar to the `blobName` parameter of the `KosRomFsMount()` function). After mounting, you can close the handle of the blob.

The `KosRomFsMount()` and `KosRomFsMountByHandle()` functions mount only in the calling process. The same ROMFS image can have several simultaneously existing mount points that were created by calls of the `KosRomFsMount()` or `KosRomFsMountByHandle()` function in one or more processes.

Opening a file

To open a file, you must call the `KosRomFsOpen()` function while specifying a string in the format `<mount point name>/<file name>` via the `path` parameter.

The same file can have several simultaneously existing identifiers that were obtained by calling the `KosRomFsOpen()` function in one or more processes. In this case, operations with the file via different identifiers are performed without mutual influence.

Reading data from a file

To read data from a file, use the `KosRomFsRead()` function. This function reads a file starting from the defined offset relative to the beginning of the file.

Receiving information about a file

To get the file size, call the `KosRomFsGetFileSize()` function.

To obtain information that includes the file size, the unique ID of the file in ROMFS, and the SID for the blob containing the ROMFS image, call the `KosRomFsGetFileStat()` function.

Closing a file

To close the file, call the `KosRomFsClose()` function.

Unmounting a ROMFS image

To unmount a ROMFS image, call the `KosRomFsUnmount()` function.

To unmount all ROMFS images, call the `KosRomFsUnmountAll()` function.

The `KosRomFsUnmount()` and `KosRomFsUnmountAll()` functions unmount only in the calling process.

A file that is open when the ROMFS image is unmounted remains available for operations until it is closed.

Information about API functions

Romfs.h functions

Function	Information about the function
`KosRomFsMount()`	Purpose Mounts a ROMFS image. Parameters • [in] `romFsName` – pointer to the name of the mount point. • [in] `blobName` – pointer to the name of the blob containing the ROMFS image. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosRomFsMountByHandle()`	Purpose Mounts a ROMFS image. Parameters • [in] `romFsName` – pointer to the name of the mount point. • [in] `blob` – handle of the blob containing the ROMFS image. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosRomFsUnmount()`	Purpose Unmounts a ROMFS image. Parameters • [in] `romFsName` – pointer to the name of the mount point. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KosRomFsUnmountAll()`	Purpose Unmounts all ROMFS images. Parameters N/A Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosRomFsOpen()`	Purpose Opens a file. Parameters • [in] `path` – pointer to a string in the format `<mount point name>/<file name>`. • [out] `file` – pointer to the file ID. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosRomFsClose()`	Purpose Closes a file. Parameters • [in] `file` – file ID. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosRomFsRead()`	Purpose Reads data from a file, starting from the defined offset. Parameters • [in] `file` – file ID. • [in] `offset` – offset (relative to the beginning of the file) from which reading should start, in bytes. • [out] `buf` – pointer to the data read buffer. • [in] `bufSize` – size of the buffer defined via the `buf` parameter, in bytes. • [out] `read` – actual size of the read data in bytes. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosRomFsGetFileSize()`	Purpose Gets the size of a file. Parameters • [in] `file` – file ID. • [out] `size` – pointer to the file size (in bytes). Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KosRomFsGetFileStat()`	Purpose Gets information about the file. Parameters • [in] `file` – file ID. • [out] `stat` – pointer to the structure containing the file size in bytes, the unique file identifier in ROMFS, and the SID for the blob containing the ROMFS image. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Using the ROMFS file system (high-level API fs.h)

The API is defined in the header file `sysroot-*-kos/include/kos/fs.h` from the KasperskyOS SDK.

This API is intended for working with the ROMFS file system that is managed by the KasperskyOS kernel or by the system program `FSUsr` (but not VFS).

> The fs.h API is obsolete, therefore you are advised to use the romfs.h API instead.

Information about API functions is provided in the table below.

> The `libkos` library also provides a low-level API for using the ROMFS file system. The low-level API should be used only if the high-level API does not have sufficient capabilities.

Selecting a file system endpoint

A process can use the ROMFS file system endpoint provided by the kernel or by the system program `FSUsr`. For more details, refer to Using the ROMFS file system (low-level API fs_api.h).

Opening a file

To open a file, call the `KosOpenFile()` function while specifying the file name via the `fname` parameter.

Reading data from a file

To read data from a file, use the `KosLoadFileToBuf()`, `KosReadFileRaw()` or `KosReadFile()` function.

The `KosLoadFileToBuf()` uses the input parameter `path` to get the pointer to the file name, therefore you do not need to open the file in advance. This function creates a buffer and copies the entire file to it. After it is used, this buffer must be flushed by calling the `KosMemFree()` function from the alloc.h API.

The `KosReadFileRaw()` function reads a file starting from the defined offset relative to the beginning of the file. The `KosReadFile()` function reads a file starting at the current location of the cursor. Immediately after a file is opened, the cursor has a zero offset relative to the beginning of the file. The `KosReadFileRaw()` function ignores and preserves the file position indicator and does not change it. A repeated call of the `KosReadFileRaw()` function with the same parameters will read from the same part of the file. The `KosReadFile()` function takes into account the file position indicator and offsets it to the end of the file according to the size of the read data. A repeated call of the `KosReadFile()` function with the same parameters will then read from a different part of the file that follows the part that was read by the previous call.

To get the position of the cursor (offset of the cursor relative to the beginning of the file), you must call the `KosGetFilePos()` function.

To set the necessary position of the cursor, you must call the `KosSeekFile()` function. The base for setting the cursor position can be the beginning of the file, the current position of the cursor, or the end of the file. If the beginning of the file is used as the base, you must specify a positive offset. If the current position of the cursor is used as the base, you must specify a positive or negative offset. If the end of the file is used as the base, you must specify a negative offset. The necessary cursor position must not go beyond the file boundaries.

Getting the file size

To get the file size, call the `KosGetFileSize()` function.

Closing a file

To close the file, call the `KosCloseFile()` function.

Information about API functions

fs.h functions

Function	Information about the function
`KosOpenFile()`	Purpose Opens a file. Parameters • [in] `fname` – pointer to the file name. Returned values: If successful, the function returns the file ID, otherwise it returns `RTL_NULL`.

Function	Information about the function
`KosReadFileRaw()`	Purpose Reads data from a file, starting from the defined offset. Parameters • [in] `file` – file ID. • [out] `buffer` – pointer to the data read buffer. • [in] `size` – size of the buffer defined via the `buffer` parameter (in bytes). • [in] `offset` – offset (relative to the beginning of the file) from which reading should start, in bytes. Returned values: If successful, the function returns the size (bytes) of the read data, otherwise it returns `0`.
`KosReadFile()`	Purpose Reads data from a file, starting with the current position of the cursor. Parameters • [in] `file` – file ID. • [out] `buffer` – pointer to the data read buffer. • [in] `size` – size of the buffer defined via the `buffer` parameter (in bytes). Returned values: If successful, the function returns the size (bytes) of the read data, otherwise it returns `0`.
`KosGetFileSize()`	Purpose Gets the size of a file. Parameters • [in] `file` – file ID. Returned values: If successful, the function returns the file size (in bytes). If it returns `0`, this indicates that either the file size is zero or an error occurred.
`KosGetFilePos()`	Purpose Gets the position of the cursor. Parameters • [in] `file` – file ID. Returned values: Cursor position (offset of the cursor relative to the beginning of the file), in bytes.

Function	Information about the function
`KosSeekFile()`	Purpose Sets the position of the cursor. Parameters • [in] `file` – file ID. • [in] `offset` – offset of the cursor relative to the base defined in the `origin` parameter (in bytes). • [in] `origin` – base for setting the cursor position: beginning of the file (`SEEK_SET`), current position of the cursor (`SEEK_CUR`), and end of the file (`SEEK_END`). Returned values: If successful, the function returns the cursor position (offset of the cursor relative to the beginning of the file), in bytes. If it returns `0`, this indicates that either a zero offset was set for the cursor or there was an error in which the cursor position defined by the `offset` and `origin` parameters is outside of the file boundaries. If this type of error occurs, the current position of the cursor is not changed.
`KosCloseFile()`	Purpose Closes a file. Parameters • [in] `file` – file ID. Returned values N/A
`KosLoadFileToBuf()`	Purpose Creates a buffer and copies the file to it. Parameters • [in] `path` – pointer to the file name. • [out] `fileBuf` – pointer to the address of the buffer containing data from the file. • [out] `fileSize` – pointer to the file size (in bytes). Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Using the ROMFS file system (low-level API fs_api.h)

API is defined in the header file `sysroot-*-kos/include/coresrv/fs/fs_api.h` from the KasperskyOS SDK.

This API is intended for working with the ROMFS file system that is managed by the KasperskyOS kernel or by the system program `FSUsr` (but not VFS).

> The fs_api.h API is obsolete, therefore you are advised to use the romfs.h API instead.

Information about API functions is provided in the table below.

> The `libkos` library also provides a high-level API for using the ROMFS file system. The low-level API should be used only if the high-level API does not have sufficient capabilities.

Selecting a file system endpoint

A process can use the ROMFS file system endpoint provided by the kernel or by the system program `FSUsr`. In the first case, the process can access the ROMFS file system that is put into the kernel space and is managed by the kernel, respectively. In the second case, the process can access the ROMFS file system that is put into the user space and is managed by the `FSUsr` program. The `FSUsr` program lets you mount any ROMFS image that may be significantly larger than the ROMFS image in the kernel space. An image of the kernel-managed ROMFS file system is created during the solution build and it is automatically loaded into the kernel space and mounted when the solution is started.

To enable the use of a ROMFS file system that is put into the user space, you must include the `FSUsr` program in your KasperskyOS-based solution. (To check whether the `FSUsr` program is included in your KasperskyOS SDK, verify that the `sysroot-*-kos/bin/fsusr` executable file is available.) The `libkos` library serves as the client library of the `FSUsr` program and automatically creates an IPC channel to the `FSUsr` program process. There should be only one `FSUsr` program process.

The specific parent process determines which of the two ROMFS file system endpoints is used by default (for details, see Managing processes (high-level API task.h)).

To get information indicating which of the two ROMFS file system endpoints is used, call the `KosTaskGetSelfFSBackend()` function from the task.h API.

To change the ROMFS file system endpoint, call the `KnFsChangeBackend()` function. If the endpoint provided by the kernel is used by default, the first call of this function with the value `KosTaskFSBackendUsr` in the `fsBackend` parameter also dynamically creates an IPC channel to the process of the `FSUsr` program. If the endpoint provided by the `FSUsr` program is used by default, the IPC channel to the process of this program is created dynamically when the `libkos` library is initialized.

You can change the ROMFS file system endpoint multiple times during the lifetime of the process.

To mount the ROMFS image that is loaded in the user space, call the `KnFsChange()` function. This function can be applied only when using the ROMFS endpoint provided by the `FSUsr` program. If a ROMFS image has already been mounted, the `KnFsChange()` function unmounts it and mounts the new one. The virtual memory region for the ROMFS image must be reserved and committed via one call of the `KnVmAllocate()` function from the vmm_api.h API. After calling the `KnFsChange()` function, this region can be freed. The physical memory occupied by the ROMFS image will be freed after the image is unmounted. Otherwise it would remain occupied until the corresponding virtual memory is freed.

To unmount the ROMFS image without mounting a new one, call the `KnFsChange()` function with the `RTL_NULL` and `0` values in the `base` and `size` parameters, respectively.

By default, a child process may use the ROMFS file system that resides in the user space and is mounted by the parent process (for details, see Managing processes (high-level API task.h)).

The `FSUsr` program process may have multiple clients at the same time. In this case, you must ensure ROMFS image mount synchronization so that one process does not replace an image if it is needed by another process. You can also use a solution security policy, which lets you prohibit calls of the interface method of the `FSUsr` program to replace the ROMFS image.

Getting file system information

To get the size of a file system, call the `KnFsGetFsSize()` function.

To find out the number of files in a file system, call the `KnFsCountFiles()` function.

To get the names of files in a file system, use the `KnFsGetFileInfo()` function. To enumerate all files, you must increment the value in the `index` parameter beginning with zero until you receive the `rcInvalidArgument` error. In addition to the file name, this function lets you get the unique ID of a file that may be used, for example, to identify the structure containing the file metadata (similar to use of the `inode` structure in UNIX-like operating systems).

Opening a file

To open a file, call the `KnFsOpen()` function while specifying the file name via the `name` parameter.

Reading data from a file

To read data from a file, use the `KnFsRead()` function. Data is read in fixed-size blocks. The data block size (in bytes) corresponds to the value of the `FileSectorSize` constant defined in the `sysroot-*-kos/include/kl/core/FS.idl` file from the KasperskyOS SDK. To read all data from a file, you must increment the value in the `sectorNumber` parameter beginning with zero until the total size of the read data is equal to the file size, or until the size of the read data received via the `read` parameter is less than the size of a data block.

Receiving information about a file

To get the file size, call the `KnFsGetFileSize()` function.

To get the unique ID of a file, call the `KnFsGetFileId()` function.

Closing a file

To close the file, call the `KnFsClose()` function.

Information about API functions

fs_api.h functions

Function	Information about the function
`KnFsOpen()`	Purpose Opens a file. Parameters • [in] `name` – pointer to the file name. • [out] `handle` – pointer to the file handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnFsClose()`	Purpose Closes a file. Parameters • [in] `handle` – file handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnFsRead()`	Purpose Reads data from a file. Parameters • [in] `handle` – file handle. • [in] `sectorNumber` – data block number. Enumeration starts with zero. • [out] `read` – pointer to the size (in bytes) of the read data. • [out] `data` – pointer to the data read buffer. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnFsGetFileSize()`	Purpose Gets the size of a file. Parameters • [in] `handle` – file handle. • [out] `size` – pointer to the file size (in bytes). Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KnFsGetFileId()`	Purpose Gets the unique ID of a file. Parameters • [in] `handle` – file handle. • [out] `id` – pointer to the unique ID of the file. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnFsCountFiles()`	Purpose Gets the number of files in the file system. Parameters • [out] `count` – pointer to the number of files in the file system. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnFsGetFileInfo()`	Purpose Gets the name and unique ID of a file based on the file index. Parameters • [in] `index` – file index. Enumeration starts with zero. • [out] `buf` – pointer to the file name buffer. • [in] `buflen` – size (in bytes) of the file name buffer. The maximum size of a file name (in bytes), including the terminating zero, corresponds to the value of the `ROMFS_NAME_MAX` constant defined in the header file `sysroot-*-kos/include/rtl/romfs_info.h` from the KasperskyOS SDK. • [optional,out] `id` – pointer to the unique ID of the file, or `RTL_NULL` if you do not need to receive this ID. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If the value in the `index` parameter is greater than or equal to the number of files in the file system, the function returns `rcInvalidArgument`.
`KnFsChange()`	Purpose Changes the ROMFS image residing in the user space. Parameters • [in,optional] `base` – pointer to the virtual memory region containing the ROMFS image, or `RTL_NULL` if you need to unmount the utilized ROMFS image without mounting a new one. The base address of the virtual memory region containing the ROMFS image must be aligned to the memory page boundary. • [in,optional] `size` – size of the virtual memory region containing the ROMFS image defined as a multiple of the memory page size, in bytes, or `0` if you need to unmount the utilized ROMFS image without mounting a new one. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KnFsGetFsSize()`	Purpose Gets the size of the ROMFS file system. Parameters • [out] `fsSize` – pointer to the size (in bytes) of the file system. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnFsChangeBackend()`	Purpose Defines whether the ROMFS file system for a calling process is supported by the kernel or by the `FSUsr` system program. Parameters • [in] `fsBackend` – value that defines how the ROMFS file system is supported (`KosTaskFSBackendKernel` – by the kernel, `KosTaskFSBackendUsr` – by the system program `FSUsr`). The parameter type is defined in the header file `sysroot-*-kos/include/kos/task.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Preparing loadable ELF image segments for creating a process (elf_api.h)

This API is defined in the header file `sysroot-*-kos/include/coresrv/elf/elf_api.h` from the KasperskyOS SDK.

The API lets you use an ELF image to create MDL buffers containing loadable segments for subsequent mapping of these segments to the virtual memory of a new process.

Information about API functions is provided in the table below.

Using the API

The standard scenario for API usage includes the following steps:

1. Create an object that describes an ELF image by calling the `KnElfOpen()` function.

   The ELF image must be put into the buffer whose pointer is received by the `KnElfOpen()` function via the `img` parameter. If the ELF image is incorrect, the function ends with an error, and the object describing the ELF image will not be created. You can verify that the ELF image is correct before completing this step by calling the `KnElfValidate()` function.

2. Create MDL buffers containing loadable ELF image segments for subsequent mapping of these segments to the virtual memory of a new process.

   To complete this step, call the `KnElfCreateVmSegEx()` function while using the `descr` input parameter to define the object describing the ELF image.

When using the `KnElfCreateVmSegEx()` function, you must take into account that the symbol table `.symtab` and the string table `.strtab` are considered to be separate loadable segments of the ELF image. In other words, in an array of structures whose address is passed via the output parameter `segs`, and in the number of the loadable segments passed via the output parameter `segsCount`, the symbol table and string table are taken into account along with segments of the `LOAD` type. The function can create an MDL buffer containing a symbol table and an MDL buffer containing a string table for subsequent mapping to the memory of a new process, irrespective of whether or not the `.symtab` and `.strtab` sections are located in a loadable ELF image segment.

To avoid having to create an MDL buffer containing a symbol table, specify `RTL_NULL` in the `symTabIndex` parameter. To avoid having to create an MDL buffer containing a string table, specify `RTL_NULL` in the `strTabIndex` parameter. If these parameters are set to any values other than `RTL_NULL`, the MDL buffers containing the symbol table and string table will be created if the ELF image contains the `.symtab` and `.strtab` sections. If the `.symtab` or `.strtab` section is absent, the `SEG_IDX_INVALID` value is passed via the `symTabIndex` or `strTabIndex` parameter, respectively.

The `KnElfCreateVmSegEx()` function uses the `relocBase` parameter to accept the ELF image load offset. The ELF image load offset is calculated by the `KnTaskVmReserveForElf()` function from the task_api.h API. If the ELF file has the `DYN` type, the `KnTaskVmReserveForElf()` function uses the `relocBase` output parameter to pass a random offset for ASLR support, and it passes a zero offset for an ELF file of a different type. The `KnElfCreateVmSegEx()` function uses the ELF image load offset to calculate the address of the program entry point (`entry` output parameter). Also while accounting for the ELF image load offset, the `KnElfCreateVmSegEx()` function calculates the load address of each loadable segment except the symbol table `.symtab` and the string table `.strtab`. The `KnElfCreateVmSegEx()` function specifies null load addresses for the symbol table and the string table. Therefore, the actual load addresses of the symbol table and string table will be determined automatically by the virtual memory management mechanism when mapping the MDL buffers containing these tables to the virtual memory of the new process. The load addresses of segments are saved by the `KnElfCreateVmSegEx()` function in the `loadAddr` field of structures whose array address is passed via the output parameter `segs`.

3. Map MDL buffers containing loadable ELF image segments to the virtual memory of a new process.

   This step is performed when using the `KosTaskInitFromSeg()`, `KosTaskInitFromSegEx()`, or `KosTaskLoadSegments()` function from the task.h API.

4. Free up resources that were allocated during successful execution of the `KnElfCreateVmSegEx()` function.

   To complete this step, call the `KnElfFreeVmSeg()` function.

5. Delete an object that describes an ELF image by calling the `KnElfClose()` function.

   After you delete an object describing an ELF image, you can flush the buffer containing the ELF image.

The `KnElfGetLoadInfo()` function lets you get the data that is used to reserve a virtual memory region in a new process for subsequent mapping of MDL buffers with `LOAD`-type ELF image segments. This function is intended for use by the `libkos` library.

Information about API functions

elf_api.h functions

Function	Information about the function
`KnElfValidate()`	Purpose Checks whether the ELF image is correct. Parameters • [in] `img` – pointer to the buffer containing the ELF image. • [in] `imgSize` – size (in bytes) of the ELF image. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If the ELF image is incorrect, the function returns `rcInvalidFormat`.
`KnElfOpen()`	Purpose Creates an object describing an ELF image. Parameters • [in] `img` – pointer to the buffer containing the ELF image. • [in] `imgSize` – size (in bytes) of the ELF image. • [out] `descr` – pointer to the address of the object describing the ELF image. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnElfClose()`	Purpose Deletes an object describing an ELF image. Parameters • [in] `descr` – pointer to the object describing the ELF image. Returned values N/A
`KnElfGetLoadInfo()`	Purpose Gets the data that is used to reserve the virtual memory region into which the ELF image segments will be loaded. Parameters • [in] `descr` – pointer to the object describing the ELF image. • [out] `info` – pointer to the structure that will store the data used to reserve the virtual memory region into which the ELF image segments will be loaded. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KnElfCreateVmSegEx()`	Purpose Creates MDL buffers containing loadable ELF image segments for subsequent mapping of these segments to the virtual memory of a new process. Parameters • [in] `descr` – pointer to the object describing the ELF image. • [in] `relocBase` – ELF image load offset (in bytes). • [out] `entry` – pointer to the address of the program entry point. • [out] `segs` – pointer to the address of the array of structures describing the loadable ELF image segments. The type of structure is defined in the header file `sysroot-*-kos/include/coresrv/vmm/vmm_types.h` from the KasperskyOS SDK. • [out] `segsCount` – pointer to the number of loaded ELF image segments. • [out,optional] `symTabIndex` – pointer to the index of the loadable ELF image segment containing the symbol table `.symtab` or `RTL_NULL` if you do not need to create an MDL buffer containing a symbol table. • [out,optional] `symTabSize` – pointer to the size of the symbol table `.symtab` (in bytes) or `RTL_NULL` if the `symTabIndex` parameter is set to `RTL_NULL`. • [out,optional] `strTabIndex` – pointer to the index of the loadable ELF image segment containing the string table `.strtab` or `RTL_NULL` if you do not need to create an MDL buffer containing a string table. • [out,optional] `strTabSize` – pointer to the size of the string table `.strtab` (in bytes) or `RTL_NULL` if the `strTabIndex` parameter is set to `RTL_NULL`. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnElfFreeVmSeg()`	Purpose Frees up resources that were allocated during successful execution of the `KnElfCreateVmSegEx()` function. Parameters • [in] `segs` – pointer to the array of structures describing loaded ELF image segments. The type of structure is defined in the header file `sysroot-*-kos/include/coresrv/vmm/vmm_types.h` from the KasperskyOS SDK. • [in] `segsCount` – number of loaded ELF image segments. Returned values N/A

Hardware platform firmware communication (efi_api.h)

This API is defined in the header file `sysroot-*-kos/include/coresrv/efi/efi_api.h` from the KasperskyOS SDK.

This API provides the capability to use the hardware platform's firmware functions that are available through the UEFI interface after KasperskyOS is loaded. (For more information about these functions, see the "Services – Runtime Services" section of the UEFI specification.)

> The extent to which the behavior of UEFI interface functions on specific hardware corresponds to the UEFI specification is determined by the developer of this hardware.

The header file `efi_api.h` provides information about how the return codes of API functions are mapped to the return codes of the hardware platform's firmware functions.

Information about API functions is provided in the table below.

Using the API

The initial step is to call the `EfiApiInit()` function, which creates a connection between the calling process and the core endpoint that facilitates interaction with firmware via the UEFI interface.

To get or set the current time, call the `KnEfiGetTime()` or `KnEfiSetTime()` function, respectively.

> The `KnEfiGetTime()` and `KnEfiSetTime()` functions cannot be used to get and set the time with high precision because the calling thread may be evicted.

To restart the hardware platform, call the `KnEfiResetSystem()` function.

To enumerate the names of UEFI variables and the unique vendor IDs related to these variables, use the `KnEfiGetNextVariableName()` function.

To get the value of a UEFI variable, call the `KnEfiGetVariable()` function.

To create, edit, or delete a UEFI variable, call the `KnEfiSetVariable()` function. To edit attributes of a UEFI variable, you must delete this variable and create a new one with the new attributes.

To get information about the sizes of UEFI variables and the memory intended for storing these variables, use the `KnEfiQueryVariableInfo()` function.

Information about API functions

efi_api.h functions

Function	Information about the function
`KnEfiGetTime()`	Purpose Lets you get the current time from the time source on the hardware platform and the characteristics of this source. Parameters • [out] `time` – pointer to the structure containing time information. The type of structure is defined in the header file `sysroot-*-kos/include/efi/types.h` from the KasperskyOS SDK. • [out,optional] `capabilities` – pointer to the structure containing time source characteristics, or `RTL_NULL` if these characteristics are not needed. The type of structure is defined in the header file `sysroot-*-kos/include/efi/types.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KnEfiSetTime()`	Purpose Sets the current time in the time source on the hardware platform. Parameters • [in] `time` – pointer to the structure containing time information. The type of structure is defined in the header file `sysroot-*-kos/include/efi/types.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnEfiGetVariable()`	Purpose Gets the value of the UEFI variable. Parameters • [in] `variableName` – pointer to the name of the variable. The type of symbols is defined in the header file `sysroot-*-kos/include/efi/types.h` from the KasperskyOS SDK. The maximum size of a variable name (in symbols), including the terminating zero, corresponds to the value of the `MaxEfiStringLength` constant defined in the file `sysroot-*-kos/include/kl/core/Efi.idl` from the KasperskyOS SDK. • [in] `vendorGuid` – pointer to the structure containing the unique ID of the vendor related to the variable. The type of structure is defined in the header file `sysroot-*-kos/include/efi/types.h` from the KasperskyOS SDK. • [out,optional] `attributes` – pointer to the value whose bits are interpreted as the attributes of the variable, or `RTL_NULL` if the variable attributes are not needed. The constants of variable attributes are defined in the header file `sysroot-*-kos/include/efi/types.h` from the KasperskyOS SDK. • [in,out] `dataSize` – input data: pointer to the buffer size defined via the `data` parameter (in bytes), or the pointer to the null integer value to determine the required size of this buffer; output data: pointer to the size of the data stored in the buffer as defined via the `data` parameter (in bytes), or the required size of this buffer (in bytes). The maximum data size (in bytes) corresponds to the value of the `MaxEfiVariableSize` constant defined in the `sysroot-*-kos/include/kl/core/Efi.idl` file from the KasperskyOS SDK. • [out,optional] `data` – pointer to the buffer containing the variable value. You can specify `RTL_NULL` if the function accepts a null value via the `dataSize` parameter. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KnEfiGetNextVariableName()`	Purpose Enumerates the names of the UEFI variables. Parameters • [in,out] `variableNameSize` – input data: pointer to the buffer size defined via the `variableName` parameter (in bytes); output data: pointer to the required buffer size for the variable name if the function terminates with the `rcBufferTooSmall` code (in bytes). The maximum buffer size (in bytes) corresponds to the doubled value of the `MaxEfiStringLength` constant defined in the `sysroot-*-kos/include/kl/core/Efi.idl` file from the KasperskyOS SDK. • [in,out] `variableName` – input data: pointer to the buffer containing the variable name (with a terminating zero) received during the previous call of the function; output data: pointer to the buffer containing the variable name (with a terminating zero) received during the current call of the function. The type of symbols is defined in the header file `sysroot-*-kos/include/efi/types.h` from the KasperskyOS SDK. If the function is being called for the first time, you must pass a buffer filled with zeros. • [in,out] `vendorGuid` – input data: pointer to the structure containing the unique ID of the vendor that is related to the variable and whose name was received during the previous call of the function; output data: pointer to the structure containing the unique ID of the vendor that is related to the variable and whose name was received during the current call of the function. The type of structure is defined in the header file `sysroot-*-kos/include/efi/types.h` from the KasperskyOS SDK. If the function uses the `variableName` parameter to accept a buffer filled with zeros, the contents of the structure passed to the function via this parameter is ignored. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnEfiSetVariable()`	Purpose Specifies the value of the UEFI variable. Parameters • [in] `variableName` – pointer to the name of the variable. The type of symbols is defined in the header file `sysroot-*-kos/include/efi/types.h` from the KasperskyOS SDK. The maximum size of a variable name (in symbols), including the terminating zero, corresponds to the value of the `MaxEfiStringLength` constant defined in the file `sysroot-*-kos/include/kl/core/Efi.idl` from the KasperskyOS SDK. • [in] `vendorGuid` – pointer to the structure containing the unique ID of the vendor related to the variable. The type of structure is defined in the header file `sysroot-*-kos/include/efi/types.h` from the KasperskyOS SDK. • [in] `attributes` – value whose bits are interpreted as attributes of the variable. The constants of variable attributes are defined in the header file `sysroot-*-kos/include/efi/types.h` from the KasperskyOS SDK. • [in] `dataSize` – size of data in the buffer as defined via the `data` parameter (in bytes), or `0` to delete the variable. The maximum size of this data (in bytes) corresponds to the value of the `MaxEfiVariableSize` constant defined in the `sysroot-*-kos/include/kl/core/Efi.idl` file from the KasperskyOS SDK. • [in,optional] `data` – pointer to the buffer containing the variable value. You can specify `RTL_NULL` if the `dataSize` parameter is set to `0`. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KnEfiQueryVariableInfo()`	Purpose Gets information about UEFI variables. Parameters • [in] `attributes` – value whose bits are interpreted as attributes of the variables whose information you need to get. Each corresponding variable must have the entire set of defined attributes (excluding those that are ignored by the UEFI specification). The constants of variable attributes are defined in the header file `sysroot-*-kos/include/efi/types.h` from the KasperskyOS SDK. • [out] `maximumVariableStorageSize` – pointer to the size (in bytes) of memory that is intended for storing variables with specific attributes. • [out] `remainingVariableStorageSize` – pointer to the size (in bytes) of the unused portion of memory that is intended for storing variables with specific attributes. • [out] `maximumVariableSize` – pointer to the maximum size among all the sizes of variables with the defined attributes (in bytes). Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnEfiResetSystem()`	Purpose Restarts the hardware platform. Parameters • [in] `resetType` – value that defines the restart type, such as cold or hot restart. The parameter type is defined in the header file `sysroot-*-kos/include/efi/types.h` from the KasperskyOS SDK. • [in] `resetStatus` – code that indicates the restart status, such as normal restart or restart due to malfunction. If it is a normal restart, `EFI_SUCCESS` must be specified. The parameter type and error codes are defined in the header file `sysroot-*-kos/include/efi/errors.h` from the KasperskyOS SDK. • [in,optional] `dataSize` – size of data in the buffer as defined via the `resetData` parameter (in bytes), or `0` if you do not need to specify any information about the causes of the restart. The maximum size of this data (in bytes) corresponds to the value of the `MaxEfiResetDataSize` constant defined in the `sysroot-*-kos/include/kl/core/Efi.idl` file from the KasperskyOS SDK. • [in,optional] `resetData` – pointer to the buffer containing information about the causes of the restart. You can specify `RTL_NULL` if the `dataSize` parameter is set to `0`. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`EfiApiInit()`	Purpose Creates a connection between the calling process and the core endpoint that facilitates interaction with the hardware platform's firmware via the UEFI interface. Parameters N/A Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Converting UUID and GUID presentation formats (uuid.h, guid.h, uuid_common.h)

APIs are defined in the following header files from the KasperskyOS SDK:

- `sysroot-*-kos/include/kos/uuid.h`
- `sysroot-*-kos/include/kos/guid.h`
- `sysroot-*-kos/include/kos/uuid_common.h`

The `uuid.h` and `guid.h` APIs let you convert the presentation format for UUIDs (for details about UUID, see RFC 4122) and GUIDs (Microsoft implementation of UUID), respectively. A UUID and GUID may be presented as a byte array, a formatted string (example: `2638179e-9732-4247-aa42-bd86ecc7bde0`), or a structure. The functions of the `uuid.h` and `guid.h` APIs perform the conversion between these formats. Both the UUID and GUID are a 16-byte ID consisting of five segments. The difference between a UUID and GUID is that the contents of all segments of a UUID reside in memory in big-endian, while the contents of the first three segments of a GUID reside in memory in little-endian and only the contents of the last two segments reside in memory in big-endian.

The `uuid_common.h` API contains additional macros and a function. This function lets you verify that a formatted string containing UUID or GUID data is correct, and macros can be applied when performing formatted output of a UUID or GUID using functions of the `printf` family.

Information about API functions and macros is provided in the tables below.

uuid.h functions

Function	Information about the function
`UuidStructToRaw()`	Purpose Copies UUID data from a structure to a byte array. Parameters • [in] `uuid` – pointer to the structure containing UUID data. The structure type is defined in the header file `sysroot-*-kos/include/kos/uuid_common.h` from the KasperskyOS SDK. • [out] `uuidRaw` – ID of the byte array containing UUID data. The parameter type is defined in the header file `sysroot-*-kos/include/kos/uuid_common.h` from the KasperskyOS SDK. Returned values N/A
`UuidRawToStruct()`	Purpose Copies UUID data from a byte array to a structure. Parameters • [in] `uuidRaw` – ID of the byte array containing UUID data. The parameter type is defined in the header file `sysroot-*-kos/include/kos/uuid_common.h` from the KasperskyOS SDK. • [out] `uuid` – pointer to the structure containing UUID data. The structure type is defined in the header file `sysroot-*-kos/include/kos/uuid_common.h` from the KasperskyOS SDK. Returned values N/A

Function	Information about the function
`UuidStringToRaw()`	Purpose Copies UUID data from a formatted string to a byte array. Parameters • [in] `uuidStr` – pointer to the formatted string containing UUID data. • [out] `uuidRaw` – ID of the byte array containing UUID data. The parameter type is defined in the header file `sysroot-*-kos/include/kos/uuid_common.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If the formatted string containing UUID data is incorrect (has an inappropriate size, incorrect hyphen placement or invalid symbols), the function returns `rcInvalidArgument`.
`UuidRawToString()`	Purpose Copies UUID data from a byte array to a formatted string. Parameters • [in] `uuidRaw` – ID of the byte array containing UUID data. The parameter type is defined in the header file `sysroot-*-kos/include/kos/uuid_common.h` from the KasperskyOS SDK. • [out] `uuidStr` – pointer to the formatted string containing UUID data. The string size (in bytes) including the terminating zero matches the value of the `UUID_STR_SIZE` constant defined in the header file `sysroot-*-kos/include/kos/uuid_common.h` from the KasperskyOS SDK. Returned values N/A
`UuidStringToStruct()`	Purpose Copies UUID data from a formatted string to a structure. Parameters • [in] `uuidStr` – pointer to the formatted string containing UUID data. • [out] `uuid` – pointer to the structure containing UUID data. The structure type is defined in the header file `sysroot-*-kos/include/kos/uuid_common.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If the formatted string containing UUID data is incorrect (has an inappropriate size, incorrect hyphen placement or invalid symbols), the function returns `rcInvalidArgument`.
`UuidStructToString()`	Purpose Copies UUID data from a structure to a formatted string. Parameters • [in] `uuid` – pointer to the structure containing UUID data. The structure type is defined in the header file `sysroot-*-kos/include/kos/uuid_common.h` from the KasperskyOS SDK. • [out] `uuidStr` – pointer to the formatted string containing UUID data. The string size (in bytes) including the terminating zero matches the value of the `UUID_STR_SIZE` constant defined in the header file `sysroot-*-kos/include/kos/uuid_common.h` from the KasperskyOS SDK. Returned values N/A

guid.h functions

Function	Information about the function
`GuidStructToRaw()`	Purpose Copies GUID data from a structure to a byte array. Parameters • [in] `guid` – pointer to the structure containing GUID data. The structure type is defined in the header file `sysroot-*-kos/include/kos/uuid_common.h` from the KasperskyOS SDK. • [out] `guidRaw` – ID of the byte array containing GUID data. The parameter type is defined in the header file `sysroot-*-kos/include/kos/uuid_common.h` from the KasperskyOS SDK. Returned values N/A
`GuidRawToStruct()`	Purpose Copies GUID data from a byte array to a structure. Parameters • [in] `guidRaw` – ID of the byte array containing GUID data. The parameter type is defined in the header file `sysroot-*-kos/include/kos/uuid_common.h` from the KasperskyOS SDK. • [out] `guid` – pointer to the structure containing GUID data. The structure type is defined in the header file `sysroot-*-kos/include/kos/uuid_common.h` from the KasperskyOS SDK. Returned values N/A
`GuidStringToRaw()`	Purpose Copies GUID data from a formatted string to a byte array. Parameters • [in] `guidStr` – pointer to the formatted string containing GUID data. • [out] `guidRaw` – ID of the byte array containing GUID data. The parameter type is defined in the header file `sysroot-*-kos/include/kos/uuid_common.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If the formatted string containing GUID data is incorrect (has an inappropriate size, incorrect hyphen placement or invalid symbols), the function returns `rcInvalidArgument`.
`GuidRawToString()`	Purpose Copies GUID data from a byte array to a formatted string. Parameters • [in] `guidRaw` – ID of the byte array containing GUID data. The parameter type is defined in the header file `sysroot-*-kos/include/kos/uuid_common.h` from the KasperskyOS SDK. • [out] `guidStr` – pointer to the formatted string containing GUID data. The string size (in bytes) including the terminating zero matches the value of the `UUID_STR_SIZE` constant defined in the header file `sysroot-*-kos/include/kos/uuid_common.h` from the KasperskyOS SDK. Returned values N/A

Function	Information about the function
`GuidStringToStruct()`	Purpose Copies GUID data from a formatted string to a structure. Parameters • [in] `guidStr` – pointer to the formatted string containing GUID data. • [out] `guid` – pointer to the structure containing GUID data. The structure type is defined in the header file `sysroot-*-kos/include/kos/uuid_common.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If the formatted string containing GUID data is incorrect (has an inappropriate size, incorrect hyphen placement or invalid symbols), the function returns `rcInvalidArgument`.
`GuidStructToString()`	Purpose Copies GUID data from a structure to a formatted string. Parameters • [in] `guid` – pointer to the structure containing GUID data. The structure type is defined in the header file `sysroot-*-kos/include/kos/uuid_common.h` from the KasperskyOS SDK. • [out] `guidStr` – pointer to the formatted string containing GUID data. The string size (in bytes) including the terminating zero matches the value of the `UUID_STR_SIZE` constant defined in the header file `sysroot-*-kos/include/kos/uuid_common.h` from the KasperskyOS SDK. Returned values N/A

uuid_common.h functions and macros

Function/macro	Information about the function/macro
`UUID_PRINT_FMT`	Purpose Expands into a formatting string for a function of the `printf` family when performing formatted output of UUID or GUID. Parameters N/A Macro values Formatting string for a function of the `printf` family.
`UUID_PRINT_PARAMS()`	Purpose Expands into a sequence of `printf` family function parameters that match the formatting string resulting from expanding the `UUID_PRINT_FMT` macro when performing formatted output of a UUID or GUID. Parameters • [in] `uuid` – pointer to the structure containing UUID or GUID data. The structure type is defined in the header file `sysroot-*-kos/include/kos/uuid_common.h` from the KasperskyOS SDK. Macro values Sequence of `printf` family function parameters that match the formatting string resulting from expanding the `UUID_PRINT_FMT` macro.

Function/macro	Information about the function/macro
`UuidValidateString()`	Purpose Checks whether a formatted string containing UUID or GUID data is correct (has an appropriate size, correct hyphen placement, and valid symbols). Parameters • [in] `uuidStr` – pointer to the formatted string containing UUID or GUID data. Returned values: If the verification is successful, the function returns `rcOk`, otherwise it returns `rcInvalidArgument`.

Asynchronous formatted output (log.h)

The API is defined in the header file `sysroot-*-kos/include/kos/log.h` from the KasperskyOS SDK.

The API enables formatted output similar to functions of the `printf` family but asynchronously, meaning while executing non-blocking calls. For example, this provides the capability for formatted output in interrupt handlers in which you could not otherwise execute calls that are blocking calls and therefore have an indeterminate duration.

Information about API functions is provided in the table below.

Using the API

The standard scenario for API usage includes the following steps:

1. Create a log.

   To complete this step, call the `KosLogCreateEx()` or `KosLogCreate()` function. A log is an object that contains a queue of formatted strings that are organized according to the FIFO (First In, First Out) principle. Creating a log automatically creates a thread that runs a loop to extract the first string from the queue in this log and perform one of the following actions:

   - Puts the string extracted from the log into the diagnostic output that is written, for example, to a COM port or USB port (version 3.0 or later, with DbC support) depending on the kernel configuration.

     This variant is applied if the log was created by calling the `KosLogCreateEx()` function with the `RTL_NULL` value in the `logFunc` parameter or by calling the `KosLogCreate()` function.

   - It passes the string extracted from the log to the callback function defined via the `logFunc` parameter of the `KosLogCreateEx()` function.

     The callback function receives the pointer to the formatted string and the size of this string (without taking into account the terminating zero, in bytes) via its own `buf` and `len` parameters, respectively. For example, this callback function may write the formatted string to the buffer instead of sending it to diagnostic output.

   When using the `KosLogCreate()` function, the maximum number of formatted strings in a log and their maximum size (taking into account the terminating zero, in bytes) are defined by the `KOS_LOG_QUEUE_CAPACITY_DEFAULT` and `KOS_LOG_MSG_SIZE_DEFAULT` constants, respectively. The `KosLogCreateEx()` function lets you define these log parameters.

2. Print formatted output.

To print a formatted string, you must add this string to the log by calling the `KosLogVWrite()` or `KosLogWrite()` function. Optionally, the `KosLogWrite()` function receives a sequence of parameters (`...`) whose values must be put into the formatted string. The `KosLogVWrite()` function receives the list of values to put into the formatted string via the `ap` parameter of the `rtl_va_list` type. Therefore, this function must be used together with the `rtl_va_start()` and `rtl_va_end()` macros defined in the header file `sysroot-*-kos/include/rtl/stdarg.h` from the KasperskyOS SDK.

If the log is completed filled, the `KosLogVWrite()` and `KosLogWrite()` functions end with the `rcOutOfMemory` error. In addition, the thread that extracts the formatted strings from the log puts a message about data loss into the diagnostic output or uses another method to pass this message (depending on the implementation of the callback function defined via the `logFunc` parameter of the `KosLogCreateEx()` function).

3. Delete the log.

   To complete this step, call the `KosLogDestroy()` function.

   Prior to deletion, you can verify that the log is empty by calling the `KosLogIsEmpty()` function.

Information about API functions

log.h functions

Function	Information about the function
`KosLogCreateEx()`	Purpose Creates a log. Parameters • [in] `msgSize` – maximum size of formatted strings in the log, taking into account the terminating zero (in bytes). It must not exceed the value of the `KOS_LOG_MSG_SIZE_MAX` constant. • [in] `capacity` – maximum number of formatted strings in the log. It must not exceed the value of the `KOS_LOG_QUEUE_CAPACITY_MAX` constant. • [in,optional] `logFunc` – ID of the callback function that is called when each formatted string is extracted from the log, or `RTL_NULL` to put the formatted strings extracted from the log into diagnostic output. • [out] `outH` – pointer to the log ID. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosLogCreate()`	Purpose Creates a log with the default parameters. Parameters • [out] `outH` – pointer to the log ID. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KosLogDestroy()`	Purpose Deletes a log. Parameters • [in] `handle` – log identifier. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosLogVWrite()`	Purpose Adds a formatted string to a log. Parameters • [in] `handle` – log identifier. • [in] `fmt` – pointer to the formatting string (just like for functions of the `printf` family). • [in] `ap` – list of parameters whose values must be put into the formatted string. The list type is defined in the header file `sysroot-*-kos/include/rtl/stdarg.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information Thread-safe function. Non-blocking call.
`KosLogWrite()`	Purpose Adds a formatted string to a log. Parameters • [in] `handle` – log identifier. • [in] `fmt` – pointer to the formatting string (just like for functions of the `printf` family). • [in,optional] `...` – sequence of parameters whose values must be put into the formatted string. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information Thread-safe function. Non-blocking call.
`KosLogIsEmpty()`	Purpose Verifies whether a log is empty. Parameters • [in] `handle` – log identifier. Returned values: If the verification is successful, the function returns `rtl_true`, otherwise it returns `rtl_false`.

Multi-step thread-safe initialization and finalization (init_fini.h)

The API is defined in the header file `sysroot-*-kos/include/kos/init_fini.h` from the KasperskyOS SDK.

The API lets you define a sequence of initialization/finalization steps that will be performed in their forward order during initialization and in reverse order during finalization, for example, when a program starts and terminates. This sequence of steps is an array of structures that, along with other fields, contain fields for the IDs of callback functions for initialization and finalization. In other words, a step is a structure that contains the ID of the callback function for initialization and/or the ID of the callback function for finalization. Initialization occurs via calls of the initialization callback functions when the steps are enumerated in ascending order of their indexes in the array. Finalization occurs via calls of the finalization callback functions when the steps are enumerated in descending order of their indexes in the array. If an error occurs at step *i* when calling the initialization callback function, initialization does not continue. Instead, finalization is performed starting with step *i-1* to undo the part of initialization that was successfully completed prior to the error. If an error occurs when calling the finalization callback function, finalization does not continue.

The startup of initialization and finalization has a thread-safe implementation.

Information about API macros is provided in the table below.

Creating a sequence of initialization/finalization steps

To create a sequence of initialization/finalization steps, use the `KOS_STEPS_COUNT_DECL_BEGIN()`, `KOS_APPEND_STEP_EX()`/`KOS_APPEND_STEP()`, and `KOS_STEPS_DECL_END` macros.

Example:

```
/* Create a sequence of initialization/finalization steps named Client.
 * This name must be unique within a single translation unit.
KOS_STEPS_COUNT_DECL_BEGIN(Client)
KOS_APPEND_STEP_EX("Transport", RTL_NULL, TransportInit, RTL_NULL, RTL_NULL, 0)
KOS_APPEND_STEP("Client objects", ClientObjInit, ClientObjFini)
KOS_APPEND_STEP("Group subscription objects", GroupSubscrObjInit, GroupSubscrObjFini)
KOS_APPEND_STEP("Group publication objects", GroupPubObjInit, GroupPubObjFini)
KOS_APPEND_STEP("Initialization flag", InitializationFlagSet, InitializationFlagReset)
KOS_STEPS_DECL_END
```

> In the source code file, the construct for creating the sequence of initialization/finalization steps must be placed into a global scope area (outside of the definitions of functions) where there is access to the initialization and finalization callback functions and access to the data that will be passed to these callback functions via the `_init_ctx_` and `_fini_ctx_` parameters of the `KOS_APPEND_STEP_EX()` macro.

The `KOS_STEPS_COUNT_DECL_BEGIN()` and `KOS_STEPS_DECL_END` macros declare the beginning and end of the sequence of initialization/finalization steps, respectively.

The `KOS_APPEND_STEP_EX()` and `KOS_APPEND_STEP()` macros declare the initialization/finalization step. You can use these macros to add a step to the sequence in which initialization and symmetric finalization will be performed. For this step, specify the ID of the initialization callback function in the `_init_cb_` parameter and specify the ID of the symmetric finalization callback function in the `_fini_cb_` parameter. For example, you can also add a step to the sequence in which only finalization will be performed. For this step, set the `_init_cb_` parameter to `RTL_NULL`, and set the `_fini_cb_` parameter to the ID of a finalization callback function that is symmetric in relation to the several initialization callback functions defined by the subsequent declarations of steps.

The `KOS_APPEND_STEP_EX()` macro is an extended version of the `KOS_APPEND_STEP()` macro that lets you define not only the initialization and finalization callback functions (the `_init_cb_` and `_fini_cb_` parameters) but also the data that will be passed to these callback functions (the `_init_ctx_` and `_fini_ctx_` parameters). The `KOS_APPEND_STEP_EX()` macro also lets you indicate whether you need to log errors of the initialization callback function at a declared step (the `_err_verbosity_` parameter). When using the `KOS_APPEND_STEP()` macro, errors of the initialization callback function at the declared step are always logged.

You can use the `_step_name_` parameter of the `KOS_APPEND_STEP_EX()` and `KOS_APPEND_STEP()` macros to define the name of the initialization/finalization step so that the error logging data contains the name of this step. If you don't do this, the declared step in the error logging data can be identified only by its number. The numbering of steps begins with zero and matches the order in which the steps are declared. In the example, the step named `Transport` has the number 0, the step named `Client objects` has the number 1, and the step named `Group subscription objects` has the number 2.

The initialization and finalization callback functions for the `KOS_APPEND_STEP_EX()` and `KOS_APPEND_STEP()` macros have different signatures. The signature of the initialization callback function for the `KOS_APPEND_STEP_EX()` macro corresponds to the `KosInitItInitCallback` type. This callback function uses its own `context` parameter to receive arbitrary data defined via the `_init_ctx_` parameter of the `KOS_APPEND_STEP_EX()` macro. If successful, the function returns `rcOk`, otherwise it returns an error code. The signature of the finalization callback function for the `KOS_APPEND_STEP_EX()` macro corresponds to the `KosInitItFiniCallback` type. This callback function uses its own `context` parameter to receive arbitrary data defined via the `_fini_ctx_` parameter of the `KOS_APPEND_STEP_EX()` macro, and uses its `stepType` parameter to receive the finalization type (`KosInitItStepFini` refers to normal finalization, while `KosInitItStepRollBack` refers to finalization resulting from an error of the initialization callback function or canceled initialization). If successful, the function returns `rcOk`, otherwise it returns an error code. The signature of the initialization and finalization callback functions for the `KOS_APPEND_STEP()` macro corresponds to the `KosInitItStepCallback` type. These callback functions do not have any parameters. If successful, the functions return `rcOk`, otherwise they return an error code.

Canceling initialization

You can use the `KOS_SET_INIT_CANCEL_CALLBACK()` macro to define an initialization cancellation callback function for a sequence of initialization/finalization steps. The signature of this callback function corresponds to the `KosInitItCancelCallback` type. The initialization cancellation callback function is called at each step during initialization and receives the number of the specific step via the `stepIndx` parameter. If the initialization cancellation callback function returns `rtl_true` at step *i*, the initialization callback function corresponding to step *i* is not called. Instead, finalization is performed starting with step *i-1*. If the initialization cancellation callback function returns `rtl_false` at step *i*, the initialization callback function corresponding to step *i* is called and then proceeds to step *i+1* if the initialization callback function is successfully completed.

You can use the initialization cancellation callback function to control the fulfillment of certain conditions during initialization. For example, you can monitor the time it takes to complete steps and cancel initialization if a specific duration is exceeded. You can also use this callback function to perform certain preparatory operations before calling the initialization callback function at a specific step.

The `KOS_SET_INIT_CANCEL_CALLBACK()` macro has a value consisting of the ID of the preceding initialization cancellation callback function. By saving this value, you will have the capability to subsequently use the preceding initialization cancellation callback function.

> In the source code file, the definitions of functions that use the `KOS_SET_INIT_CANCEL_CALLBACK()` macro must be positioned after the construct for creating the sequence of initialization/finalization steps.

Logging errors

If the initialization or finalization callback function at a step ends with an error, a message about this error is put into diagnostic output by default. (When using the `KOS_APPEND_STEP_EX()` macro, you can cancel error logging of the initialization callback function at a declared step.) This message contains the name of the sequence of initialization/finalization steps, the name or number of the step (depending on whether the name was defined when declaring the step), the type of procedure (`init` refers to initialization, `fini` refers to normal finalization, and `rollback` refers to finalization resulting from an error of the initialization callback function or canceled initialization), and information about the error.

Instead of the default method, you can use your own error logging method. To do so, use the `KOS_SET_LOGGER_CALLBACK()` macro to define your own error logging callback function for a sequence of initialization/finalization steps. This callback function is called if the initialization or finalization callback function at a step ends with an error. (When using the `KOS_APPEND_STEP_EX()` macro, you can cancel error logging of the initialization callback function at a declared step.) The signature of the error logging callback function corresponds to the `KosInitItErrorLoggerCallback` type. The error logging callback function receives the name of the initialization/finalization sequence (the `subsystemName` parameter), the step number (the `failedStep` parameter), step name (if it was defined via the `stepName` parameter when declaring the step), error code (the `rc` parameter) and the type of procedure (the `stepType` parameter, which receives the following values: `KosInitItStepInit` refers to initialization, `KosInitItStepFini` refers to normal finalization, and `KosInitItStepRollBack` refers to finalization resulting from an error of the initialization callback function or canceled initialization). It does not return any values. For example, an implementation of this callback function may write error information to the buffer instead of putting it into diagnostic output.

The `KOS_SET_LOGGER_CALLBACK()` macro has a value consisting of the ID of the preceding error logging callback function. By saving this value, you will have the capability to subsequently use the preceding error logging callback function.

> In the source code file, the definitions of functions that use the `KOS_SET_LOGGER_CALLBACK()` macro must be positioned after the construct for creating the sequence of initialization/finalization steps.

Starting initialization and finalization

To start initialization according to a defined sequence of initialization/finalization steps, use the `KOS_RUN_INIT_STEPS()` macro. If the initialization callback functions at all steps are completed successfully, the macro returns the value `rcOk`. If an initialization callback function at one of the steps ended with an error, the macro returns the code of this error. If initialization is canceled, the macro returns the value `rcAbort`.

To start finalization according to a defined sequence of initialization/finalization steps, use the `KOS_RUN_FINI_STEPS()` macro. If the finalization callback functions at all steps are completed successfully, the macro returns the value `rcOk`. If a finalization callback function at one of the steps ended with an error, the macro returns the code of this error.

Example of starting initialization and finalization:

```
int main(void)
{
    int ret = -1;

    INFO(UARTSR, "UART server start");

    if (KOS_RUN_INIT_STEPS(UART) == rcOk)
    {
        KosEventWait(&shutdownEvent, rtl_false);
        KOS_RUN_FINI_STEPS(UART);
        ret = 0;
    }

    INFO(UARTSR, "UART server stop");

    return ret;
}
```

Although initialization based on a single sequence of initialization/finalization steps can be started in multiple parallel threads, essentially initialization will be performed only once. This is also true for finalization. However, you must ensure that the number of successful startups of initialization (when the `KOS_RUN_INIT_STEPS()` macro has the `rcOk` value) is equal to the number of successful startups of finalization (when the `KOS_RUN_FINI_STEPS()` macro has the value `rcOk`). This is necessary because the first successful startup of initialization actually performs the initialization and increments the initialization startup counter, while subsequent successful startups of initialization only increment this counter. However, a successful startup of finalization decrements the initialization startup counter and does not actually perform finalization until this counter takes the value that it had before the first successful startup of initialization.

> In the source code, the definitions of functions that use the `KOS_RUN_INIT_STEPS()` and `KOS_RUN_FINI_STEPS()` macros must be positioned after the construct for creating the sequence of initialization/finalization steps.

Information about API macros

init_fini.h macros

Macro	Macro details
`KOS_STEPS_COUNT_DECL_BEGIN()`	Purpose Declares the start of the sequence of initialization/finalization steps. Parameters • [in] `_name_` – name of the sequence of initialization/finalization steps. It must be unique within a single translation unit. Macro values N/A

Macro	Macro details
KOS_APPEND_STEP_EX()	Purpose Declares the step in the sequence of initialization/finalization steps. Parameters • [in,optional] _step_name_ – pointer to the string containing the name of the initialization/finalization step to make sure that the error logging data contains the name of this step, or RTL_NULL if this is not required. • [in,optional] _init_ctx_ – pointer to the data that will be passed to the initialization callback function that was defined via the _init_cb_ parameter. You can specify RTL_NULL if data does not need to be passed. The initialization callback function will receive the passed data through its own context parameter. • [in,optional] _init_cb_ – ID of the initialization callback function (the KosInitItInitCallback type) or RTL_NULL if initialization is not performed at the step. • [in,optional] _fini_ctx_ – pointer to the data that will be passed to the finalization callback function that was defined via the _fini_cb_ parameter. You can specify RTL_NULL if data does not need to be passed. The finalization callback function will receive the passed data through its own context parameter. • [in,optional] _fini_cb_ – ID of the finalization callback function (the KosInitItFiniCallback type) or RTL_NULL if finalization is not performed at the step. • [in] _err_verbosity_ – value that indicates whether error logging of the initialization callback function must be performed (0 – do not perform error logging, 1 – perform error logging). Macro values N/A
KOS_STEPS_DECL_END	Purpose Declares the end of the sequence of initialization/finalization steps. Parameters N/A Macro values N/A
KOS_APPEND_STEP()	Purpose Declares the step in the sequence of initialization/finalization steps. Parameters • [in,optional] _step_name_ – pointer to the string containing the name of the initialization/finalization step to make sure that the error logging data contains the name of this step, or RTL_NULL if this is not required. • [in,optional] _init_cb_ – ID of the initialization callback function (the KosInitItStepCallback type) or RTL_NULL if initialization is not performed at the step. • [in,optional] _fini_cb_ – ID of the finalization callback function (the KosInitItStepCallback type) or RTL_NULL if finalization is not performed at the step. Macro values N/A

Macro	Macro details
`KOS_SET_INIT_CANCEL_CALLBACK()`	Purpose Defines the initialization cancellation callback function. Parameters • [in] `_name_` – name of the sequence of initialization/finalization steps. • [in] `_cancelCb` – ID of the initialization cancellation callback function (the `KosInitItCancelCallback` type). Macro values ID of the preceding initialization cancellation callback function if one exists, otherwise `RTL_NULL`.
`KOS_SET_LOGGER_CALLBACK()`	Purpose Defines the error logging callback function. Parameters • [in] `_name_` – name of the sequence of initialization/finalization steps. • [in] `logger` – ID of the error logging callback function (the `KosInitItErrorLoggerCallback` type). Macro values ID of the preceding error logging callback function if one exists, otherwise `RTL_NULL`.
`KOS_RUN_INIT_STEPS()`	Purpose Starts initialization according to a defined sequence of initialization/finalization steps. Parameters • [in] `_name_` – name of the sequence of initialization/finalization steps. Macro values It has a value of `rcOk` if successful, otherwise it has an error code value.
`KOS_RUN_FINI_STEPS()`	Purpose Starts finalization according to a defined sequence of initialization/finalization steps. Parameters • [in] `_name_` – name of the sequence of initialization/finalization steps. Macro values It has a value of `rcOk` if successful, otherwise it has an error code value.

Using specialized loadable sections of an ELF image containing data object arrays (iterable_section.h)

The API is defined in the header file `sysroot-*-kos/include/kos/iterable_section.h` from the KasperskyOS SDK.

The API lets you put data into specialized loadable ELF image sections while building a program and gain access to this data during execution of the program. A specialized section is an array of single-type data objects, and therefore you can assume that a specialized section has a type that matches the type of data objects comprising the section. The API creates specialized sections of the required types when building a program. During execution of the program, the API provides the capability to get the number of data objects in these sections, gain access to each object based on the index, and traverse objects. It is advisable to utilize this functionality if the program may be built in multiple versions that optionally enable one or more components, such as drivers. In this case, each driver uses the API to add a data object to a specialized ELF image section. For example, this data object may be a structure containing pointers to driver initialization and finalization functions. As a result, the program build process will also create a specialized ELF image section consisting of an array of structures containing pointers to initialization and finalization functions for all drivers included in the program. During execution of the program, the API will let you initialize and finalize all drivers included in the program while traversing the data objects of this section. Building a program with a different set of drivers automatically updates this section, and the code that invokes the driver initialization and finalization functions will not require modification.

Information about API macros is provided in the table below.

Adding a data object to a specialized ELF image section

To add a data object to a specialized ELF image section, use the `KOS_ITERABLE_SECTION_REGISTER()` macro. This macro adds a data object to the specialized section of the specific type whose name is specified in the `type` parameter. The name of a specialized section type must consist of one word. To define a type whose name consists of multiple words (for example, `unsigned long`, `const volatile double`), in the `type` parameter, you must specify an alias of this type that is created using `typedef`. You can define a primitive type or a composite type.

A data object added to a specialized section normally must be initialized. This is technically not required, because you can add uninitialized data objects to the specialized section so that they can later be used together with other global variables or for the purpose of getting the number of data objects in this specialized section. However, this is not the main purpose of the API. The main purpose of the API is to put data into specialized sections when building a program and then gain access to this data during execution of the program.

Example of adding an initialized data object to a specialized section:

```
typedef struct SFooClientOps
{
    Retcode (*init)(void);
    void (*fini)(void);
} FooClientOps;

typedef struct SFooClient
{
    const FooClientOps *ops;
} FooClient;

static Retcode InitClient(void)
{
...
}

static void FiniClient(void)
{
...
}

static const FooClientOps ops = {
    .init = InitClient,
    .fini = FiniClient,
};

KOS_ITERABLE_SECTION_REGISTER(FooClient) = {
```

```
    .ops = &ops
};
```

If a translation unit in which the `KOS_ITERABLE_SECTION_REGISTER()` macro is used will be included in the program during the build, the data object will be added to the specialized section. If the required program version will not include this translation unit, the data object will not be added to the specialized section. This automatically ensures that the contents of the specialized section will comply with the required program version.

> In the source code file, the `KOS_ITERABLE_SECTION_REGISTER()` macro must be put into the global scope.

Opening access to a specialized ELF image section

To open access to a specialized ELF image section, use the `KOS_ITERABLE_SECTION_DECLARE()` macro. This macro opens access to the specialized section of the specific type whose name is specified in the `type` parameter. Opening access to a specialized section provides the capability to get the number of data objects in this specialized section, gain access to each object based on the index, and traverse objects.

The `KOS_ITERABLE_SECTION_DECLARE()` macro must be used in each translation unit in which you need to gain access to the specialized section.

> In the source code file, the `KOS_ITERABLE_SECTION_DECLARE()` macro must be put into the global scope.

Getting the number of data objects in a specialized ELF image section

To get the number of data objects in the specialized ELF image section, use the `KOS_ITERABLE_SECTION_COUNTOF()` macro. This macro has a value that corresponds to the number of data objects in the specialized section of the specific type whose name is specified in the `type` parameter.

> In the source code file, the definitions of functions that use the `KOS_ITERABLE_SECTION_COUNTOF()` macro must be positioned after the `KOS_ITERABLE_SECTION_DECLARE()` macro.

Enumerating data objects in a specialized ELF image section

To enumerate data objects in a specialized ELF image section, use the `KOS_ITERABLE_SECTION_ENUM()` macro. This macro enumerates data objects in a specialized section of the specific type whose name is specified in the `type` parameter. It has a value corresponding to the pointer to the data object with the index specified in the `index` parameter.

> In the source code file, the definitions of functions that use the `KOS_ITERABLE_SECTION_ENUM()` macro must

be positioned after the `KOS_ITERABLE_SECTION_DECLARE()` macro.

Traversing data objects in a specialized ELF image section

To traverse data objects in a specialized ELF image section, use the `KOS_ITERABLE_SECTION_FOR_EACH()` or `KOS_ITERABLE_SECTION_FOR_EACH_REVERSE()` macro. These macros traverse data objects in a specialized section of the specific type whose name is specified in the `type` parameter. The `KOS_ITERABLE_SECTION_FOR_EACH()` macro traverses objects in forward order while the `KOS_ITERABLE_SECTION_FOR_EACH_REVERSE()` macro traverses objects in reverse order. Both macros are expanded into the header of the `for` operator. The pointer whose name is defined in the `iterator` parameter will contain the address of the next data object in each iteration.

Example use of the `KOS_ITERABLE_SECTION_FOR_EACH()` macro:

```
static Retcode InitDrivers(void)
{
    Retcode rc = rcOk;

    KOS_ITERABLE_SECTION_FOR_EACH(ExampleDriverClassDriverImpl, driver)
    {
        rtl_assert(driver->name != RTL_NULL);
        rtl_assert(driver->init != RTL_NULL);
        rtl_assert(!driver->isInitialized);

        rc = driver->init();
        if (rc == rcOk)
        {
            driver->isInitialized = rtl_true;
        }
        else
        {
            ERROR(EXAMPLE_CLASS, "Failed to initialize \"%s\" driver", driver->name);
            break;
        }
    }

    if (rc != rcOk)
    {
        FiniDrivers();
    }

    return rc;
}
```

Example use of the `KOS_ITERABLE_SECTION_FOR_EACH_REVERSE()` macro:

```
static Retcode FiniDrivers(void)
{
    KOS_ITERABLE_SECTION_FOR_EACH_REVERSE(ExampleDriverClassDriverImpl, driver)
    {
        if (driver->isInitialized && driver->fini != RTL_NULL)
        {
            driver->fini();
            driver->isInitialized = rtl_false;
        }
    }
```

```
    return rcOk;
}
```

> In the source code file, the definitions of functions that use the `KOS_ITERABLE_SECTION_FOR_EACH()` and `KOS_ITERABLE_SECTION_FOR_EACH_REVERSE()` macros must be positioned after the `KOS_ITERABLE_SECTION_DECLARE()` macro.

Information about API macros

iterable_section.h macros

Macro	Macro details
`KOS_ITERABLE_SECTION_REGISTER()`	Purpose Adds a data object to a specialized section of the defined type. Parameters • [in] `type` – name of the specialized section type. Macro values N/A
`KOS_ITERABLE_SECTION_DECLARE()`	Purpose Opens access to a specialized section of the defined type. Parameters • [in] `type` – name of the specialized section type. Macro values N/A
`KOS_ITERABLE_SECTION_COUNTOF()`	Purpose Gets the number of data objects in a specialized section of the defined type. Parameters • [in] `type` – name of the specialized section type. Macro values Number of data objects in a specialized section of the defined type.
`KOS_ITERABLE_SECTION_ENUM()`	Purpose Enumerates data objects in a specialized section of the defined type. Parameters • [in] `type` – name of the specialized section type. • [in] `index` – index for enumerating data objects. Enumeration starts with zero. Macro values Pointer to the data object in the specialized section of the defined type, or `RTL_NULL` if the index is outside of the permissible range.

Macro	Macro details
`KOS_ITERABLE_SECTION_FOR_EACH()`	Purpose Traverses data objects in a specialized section of the defined type in forward order. Parameters • [in] `type` – name of the specialized section type. • [in] `iterator` – name of the pointer for saving the addresses of traversed data objects. Macro values N/A
`KOS_ITERABLE_SECTION_FOR_EACH_REVERSE()`	Purpose Traverses data objects in a specialized section of the defined type in reverse order. Parameters • [in] `type` – name of the specialized section type. • [in] `iterator` – name of the pointer for saving the addresses of traversed data objects. Macro values N/A

Logging the operation of programs (trace.h)

The API is defined in the header file `sysroot-*-kos/include/kos/trace.h` from the KasperskyOS SDK.

The API lets you forward messages about the operation of programs to the logging system, which prints diagnostic data via the UART or saves it to a file, for example. These messages are formatted strings similar to those that are generated by functions of the `printf` family. You can adjust the verbosity and log level. The verbosity level determines which API macros are expanded into code to forward messages to the logging system, and which ones are expanded into fictitious code. The higher the verbosity level, the more API macros will be activated, that is, expanded into code to forward messages to the logging system. The log level determines the maximum verbosity level for which logging is performed. In other words, messages specific to higher verbosity levels are not forwarded to the logging system. For example, if verbosity level 5 (`LOG_DEBUG`) was defined for a library, and log level 4 (`LOG_INFO`) was defined for a program that includes this library, messages from the library that are specific to verbosity level 5 (corresponding to the `DEBUG()` macro) will not be forwarded to the logging system.

Information about API macros is provided in the table below.

Using the API

Various macros of the API are intended for output of various data on program operation. The `FATAL()`, `ERROR()`, and `ERR()` macros must be used for output of error messages. The `WARN()` macro must be used for output of messages about unfavorable conditions that may subsequently lead to errors. The `INFO()` macro is provided for output of informational messages about normal operation of the program. The `DEBUG()` macro is intended for output of messages containing information for program debugging. The `TRACE()` macro must be used for output of messages containing information for detailed tracking of program operation.

Each macro accepts the following data:

- Conditional name of the program component that prints the message.

  This name will be displayed in the message header.

- Formatting string (just like for functions of the `printf` family).

- [Optional] Sequence of parameters whose values must be put into the message.

Examples:

```
TRACE(BLKPART, "Port '%s' is opened.", name);
DEBUG(USBSTORAGE, "Failed to send a reset request.");
ERR(USB, "Can't find device class %s.", name);
WARN(
    MOUNT,
    "Failed to mount rootfs with type '%s' on '%s', error: %s(%d), will retry in %u msec",
    fstab_mount.fsname,
    fstab_mount.devname,
    strerror(error),
    error,
    FSTAB_MOUNT_RETRY_DELAY_MS);
```

To set the verbosity level, you must define the `LOG_VERBOSITY` macro. This can be done in the source code file or in the `CMakeLists.txt` file.

To define the `LOG_VERBOSITY` macro, use the following values:

- `LOG_OFF` (level 0) deactivates all macros of the API.
- `LOG_FATAL` (level 1) activates the `FATAL()` macro.
- `LOG_ERR` (level 2) activates the `FATAL()`, `ERROR()`, and `ERR()` macros.
- `LOG_WARN` (level 3) activates the `FATAL()`, `ERROR()`, `ERR()`, and `WARN()` macros.
- `LOG_INFO` (level 4) activates the `FATAL()`, `ERROR()`, `ERR()`, `WARN()`, and `INFO()` macros.
- `LOG_DEBUG` (level 5) activates the `FATAL()`, `ERROR()`, `ERR()`, `WARN()`, `INFO()`, and `DEBUG()` macros.
- `LOG_TRACE` (level 6) activates all macros of the API.

Example of setting the verbosity level in the source code file:

```
#define LOG_VERBOSITY LOG_DEBUG
#include <kos/trace.h>
```

> In the source code file, you must position the directive for defining the `LOG_VERBOSITY` macro before the directive for including the header file `trace.h`. It is also advisable to refrain from including the header file `trace.h` in other header files to prevent transitive inclusion, which could lead to an obscure definition of the `LOG_VERBOSITY` macro.

To define the `LOG_VERBOSITY` macro in the `CMakeLists.txt` file, you must use the following `CMake` command:

```
target_compile_definitions(<build target name> PRIVATE LOG_VERBOSITY=<verbosity level>)
```

Example of setting the verbosity level in the `CMakeLists.txt` file:

```
target_compile_definitions(client PRIVATE LOG_VERBOSITY=5)
```

By default, verbosity levels 5 (`LOG_DEBUG`) and 4 (`LOG_INFO`) are defined for the debug and release versions of the program, respectively.

> The program build process verifies that the formatting string is compliant with the types of parameters whose values must be put into the message. This verification is performed only for activated macros of the API. Therefore, to verify all of the API macros that are used in the program source code, you must set the maximum possible verbosity level for this program.

To define the log level, you need to set the environment variable `INITIAL_LOG_LEVEL` or define the global variable `currentLogLevel` of the `int` type.

The `INITIAL_LOG_LEVEL` environment variable must be set to one of the following values:

- `CRITICAL` (level 1)
- `ERROR` (level 2)
- `WARNING` (level 3)
- `INFO` (level 4)
- `DEBUG` (level 5)
- `TRACE` (level 6)

Example of setting the logging level when packing a KPA package using the environment variable in the `RUN_CONFIGURATION->ENV` parameter of the CMake command add_kpa_component ():

```
add_kpa_component (${PACKAGE}
                   RUN_CONFIGURATION "default"
                   ...
                   ENV               "INITIAL_LOG_LEVEL=DEBUG")
```

To initialize the `currentLogLevel` variable, use the same values as you would to define the `LOG_VERBOSITY` macro, except for `LOG_OFF`. The `currentLogLevel` variable can be defined in any translation unit from the program. This will set the log level for the entire program.

Example of setting the log level in the source code file:

```
int currentLogLevel = LOG_DEBUG;
```

By default, the log level is set to 4 (`LOG_INFO`) for the debug and release versions of the program.

The `LOG_FMT` macro defines whether messages will have a header. By default, this macro has the value `LOG_FMT_ON`, which requires that messages have a header containing the following data:

- Date and time when the message was created – `[<year>-<month>-<day>T<hours>:<minutes>:<seconds>.<milliseconds>]`.
- [Only for log level 6 (`LOG_TRACE`)] Absolute path of the source code file and the number of the string that contains the API macro that forwarded the message – `[<absolute path to source code file>:<string number>]`.
- Message type – `[Fatal|Error|Warn|Info|Debug|Trace]`.
- Name of the process that sent the message – `[<process name>]`.
- Process ID (PID) and thread ID (TID) that sent the message – `[<PID>:<TID>]`.
- Conditional name of the program component that sent the message `[<program component name>]`.

Examples of messages containing a header:

```
[2025-04-16T12:54:34.191][Info][TestServer][5:5][SERVER] The server started.
[2025-04-16T13:56:12.310][/home/x/Documents/kos/kosdev/system/libs/em_transport/src/client.c:231][Trace][Einit]
[4:4][em_transport] EM transport successfully connected.
```

To remove the header from messages, you must define the `LOG_FMT` macro with the `LOG_FMT_OFF` symbolic constant value. Just like with the `LOG_VERBOSITY` macro, you must position the directive for defining the `LOG_FMT` macro before the directive for including the header file `trace.h`. You can also define the `LOG_FMT` macro as a null value by using the following `CMake` command:

```
target_compile_definitions(<build target name> PRIVATE LOG_FMT=0)
```

Information about API macros

trace.h macros

Macro	Macro details
`TRACE()`	Purpose Forwards a message containing information for detailed tracking of program operation to the logging system. Parameters • [in] `name` – program component name. • [in] `fmt` – pointer to the formatting string (just like for functions of the `printf` family). • [in,optional] `...` – sequence of parameters whose values must be put into the message. Macro values N/A

Macro	Macro details
`DEBUG()`	Purpose Forwards a message containing information for program debugging to the logging system. Parameters • [in] `name` – program component name. • [in] `fmt` – pointer to the formatting string (just like for functions of the `printf` family). • [in,optional] `...` – sequence of parameters whose values must be put into the message. Macro values N/A
`INFO()`	Purpose Forwards an informational message about normal operation of the program to the logging system. Parameters • [in] `name` – program component name. • [in] `fmt` – pointer to the formatting string (just like for functions of the `printf` family). • [in,optional] `...` – sequence of parameters whose values must be put into the message. Macro values N/A
`WARN()`	Purpose Forwards a message about unfavorable conditions that may subsequently lead to errors to the logging system. Parameters • [in] `name` – program component name. • [in] `fmt` – pointer to the formatting string (just like for functions of the `printf` family). • [in,optional] `...` – sequence of parameters whose values must be put into the message. Macro values N/A
`ERR()`	Purpose Forwards a message about an error that disrupts program operation but will not lead to abnormal termination of the program to the logging system. Parameters • [in] `...` – sequence of parameters in which the first parameter is the program component name, the second parameter is the pointer to the formatting string (just like for functions of the `printf` family), and all other optional parameters contain values that should be put into the message. Macro values N/A

Macro	Macro details
`ERROR()`	Purpose Forwards a message about an error that disrupts program operation but will not lead to abnormal termination of the program to the logging system. Parameters • [in] `name` – program component name. • [in] `fmt` – pointer to the formatting string (just like for functions of the `printf` family). • [in,optional] `...` – sequence of parameters whose values must be put into the message. Macro values N/A
`FATAL()`	Purpose Forwards a message about an error that will lead to abnormal termination of the program to the logging system. Parameters • [in] `name` – program component name. • [in] `fmt` – pointer to the formatting string (just like for functions of the `printf` family). • [in,optional] `...` – sequence of parameters whose values must be put into the message. Macro values N/A

Getting the build ID from an ELF image (elf_id.h)

The API is defined in the header file `sysroot-*-kos/include/kos/elf_id.h` from the KasperskyOS SDK.

The API lets you get the build ID from an ELF image. The build ID is required for the sampling-based profiling mechanism to work. It is also preferable that the build ID is available to the kernel GDB server that uses this ID to tell the GDB debugger where to search for debug symbols. (The kernel GDB server uses the name of the executable file as an alternative to the build ID.)

> A build ID may be absent from the ELF image. The linker adds the build ID to the ELF image only if the compiler was called with the `-Wl,--build-id` parameter.

Information about API functions is provided in the table below.

Using the API

To get the build ID and its size from an ELF image, call the `KosElfGetBuildId()` function. This data must be passed to the kernel so that it is available to the sampling-based profiling mechanism and to the GDB server of the kernel. The build ID and its size are passed to the kernel when calling the `KosTaskInitFromSeg()`, `KosTaskInitFromSegEx()` or `KosTaskLoadSegments()` function from the API task.h. These functions use the `segs` parameter to receive the array of `VmSeg` structures that describe loadable ELF image segments. The type of `VmSeg` structures is defined in the header file `sysroot-*-kos/include/coresrv/vmm/vmm_types.h` from the KasperskyOS SDK. These structures contain the `buildId` and `buildIdSize` fields. For each loadable ELF image segment (including the `.symtab` symbol table and the `.strtab` string table), you must define the build ID and its size (the same values for all segments). At the same time, you can use the `KnElfCreateVmSegEx()` function from the API elf_api.h that automatically fills the `buildId` and `buildIdSize` fields in the array of `VmSeg` structures (if the build ID is available in the ELF image) and passes the address of this array via the output parameter `segs`. In other words, when using the `KnElfCreateVmSegEx()` function, the `KosElfGetBuildId()` function is not necessary.

In addition, the `buildId` and `buildIdSize` fields of the `VmSeg` structure are automatically filled by the `KnVmSegInitFromBuf()` function from the API vmm_api.h, which receives the build ID and its size via the input parameters `buildId` and `buildIdSize`, respectively. The build ID and its size are written to the `VmSeg` structure that is defined via the output parameter `seg` so that it can then be passed to the kernel by calling the `KosTaskInitFromSeg()`, `KosTaskInitFromSegEx()` or `KosTaskLoadSegments()` function.

> The `KosTaskInit()`, `KosTaskInitEx()`, and `KosTaskLaunch()` functions from the task.h API pass the build ID and its size to the kernel if the build ID is available in the ELF image.

Information about API functions

elf_id.h functions

Function	Information about the function
`KosElfGetBuildId()`	Purpose Gets the build ID from an ELF image. Parameters • [in] `elf` – pointer to the buffer containing the ELF image. • [in] `elfSize` – size (in bytes) of the ELF image. • [in] `buildIdMaxSize` – size (in bytes) of the buffer used to store the build ID. • [out] `buildId` – pointer to the buffer used to store the build ID. • [out] `buildIdSize` – pointer to the build ID size (in bytes). Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If the build ID is absent from the ELF image, the function returns `rcResourceNotFound`.

Getting build IDs for ELF images loaded into process memory (task_deb_api.h)

The API is defined in the header file `sysroot-*-kos/include/coresrv/task/task_deb_api.h` from the KasperskyOS SDK.

The API lets you get the build IDs for ELF images (executable file and dynamic libraries) loaded into process memory. (These IDs will be received only if they are present in the KasperskyOS kernel. When a process is created or a dynamic library is loaded, the build ID is passed to the kernel only if it is present in the ELF image.) For example, build IDs can be printed together with stack backtrace data and can be used during an analysis of this data to convert memory addresses into function names and source code line numbers by using the llvm-symbolizer or llvm-addr2line tool from the KasperskyOS SDK (executable files in the `toolchain/bin` directory).

Information about API functions is provided in the table below.

Using the API

To get the build ID, call the `KnTaskGetBuildId()` function. To identify the ELF image, this function uses the `addr` parameter to get the address belonging to the memory region of the calling process that is occupied by a segment of the ELF image. You need to use this parameter to pass the memory address from the stack backtrace data, which can be obtained, for example, by calling the `backtrace()` function declared in the header file `sysroot-*-kos/include/execinfo.h` from the KasperskyOS SDK (`libc` library API).

Information about API functions

task_deb_api.h functions

Function	Information about the function
`KnTaskGetBuildId()`	Purpose Gets the build ID for the ELF image whose segment occupies a region in the memory of the calling process that includes the specified address. Parameters • [in] `addr` – address within the memory region of the process occupied by the ELF image segment. • [in] `buildIdMaxSize` – size (in bytes) of the buffer used to store the build ID. A 20-byte buffer is sufficient. • [out] `BuildId` – pointer to the buffer used to store the build ID. • [out] `BuildIdSize` – pointer to the build ID size (in bytes). Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Gaining access to named blobs (blob_api.h)

This API is defined in the header file `sysroot-*-kos/include/coresrv/blob/blob_api.h` from the KasperskyOS SDK.

The API is designed to gain access to named blobs (Binary Large Objects). You can gain access to the following blobs:

- ROMFS image provided as part of the KasperskyOS kernel image.

  Blob name: `romfs.0`.

- ROMFS image placed in memory by the bootloader (similar to initrd in Linux).

  Blob name: `romfs.1`.

Information about API functions is provided in the table below.

Getting the handle of a blob

To get the handle of a blob based on the name of this blob, call the `KnBlobGet()` function.

The BLOB handle can be transferred to another process via IPC.

The handle of the blob containing the ROMFS image is used not only by the blob_api.h API functions but is also used as input data for the `KosRomFsMountByHandle()` function from the romfs.h API.

Mapping a blob to a virtual memory region

To reserve a virtual memory region and map the BLOB to it, call the `KnBlobMap()` function.

The same blob can be mapped simultaneously to multiple virtual memory regions in one or more processes.

To free a virtual memory region that was reserved by calling the `KnBlobMap()` function, you must close the handle of this region.

Getting the size of a blob

To get the size of a blob, call the `KnBlobGetSize()` function. The size of a blob is also passed through the `size` output parameter of the `KnBlobGet()` function.

Deleting a blob

To initiate the deletion of a blob, call the `KnBlobRemove()` function. The blob will be deleted only after all of the following conditions are fulfilled:

1. Each virtual memory region to which the blob is mapped is freed in all processes that mapped it.

   When using the romfs.h API, you should keep in mind that when a ROMFS image is mounted, a call of the `KosRomFsMount()` or `KosRomFsMountByHandle()` function will reserve the virtual memory region and map a blob to it. When you unmount a ROMFS image, a call of the `KosRomFsUnmount()` or `KosRomFsUnmountAll()` function frees the virtual memory region to which the blob is mapped.

2. Each BLOB handle in all processes that own these handles is closed or revoked.

   When using the romfs.h API, you should keep in mind that the `KosRomFsMount()` function closes the blob handle while the `KosRomFsMountByHandle()` function does not close it. Therefore, after calling the `KosRomFsMountByHandle()` function, you must manually close the handle.

> Blob deletion may be prohibited by the KasperskyOS kernel. In this case, the `KnBlobRemove()` function returns `rcInvalidOperation`.

Information about API functions

Blob_api.h functions

Function	Information about the function
`KnBlobGet()`	Purpose Gets the handle of a blob based on the name of this blob. Parameters • [in] `name` – pointer to the BLOB name. • [out] `rid` – pointer to the blob handle. • [out, optional] `size` – pointer to the size of the blob in bytes, or `RTL_NULL` if the blob size is not required. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnBlobRemove()`	Purpose Initiates deletion of a blob. Parameters • [in] `name` – pointer to the BLOB name. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`KnBlobMap()`	Purpose Reserves a virtual memory region and maps the BLOB to it. Parameters • [in] `rid` – blob handle. • [in] `prot` – flags defining the access rights to the BLOB. The flags are defined in the header file `sysroot-*-kos/include/vmm/flags.h` from the KasperskyOS SDK. • [in, optional] `attr` – flags that specify the parameters of the virtual memory region, or `0` if these parameters are not required. The flags are defined in the header file `sysroot-*-kos/include/vmm/flags.h` from the KasperskyOS SDK. • [out] `addr` – pointer to the base address of the virtual memory region. • [out] `handle` – pointer to the handle of the virtual memory region. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information In the `prot` parameter, you can specify the following combinations of flags: • `VMM_FLAG_READ` – read access. • `VMM_FLAG_READ | VMM_FLAG_WRITE` – read-and-write access. In the `attr` parameter, you can specify the following flags: • `VMM_FLAG_CACHE_DISABLE` – disable caching.
`KnBlobGetSize()`	Purpose Gets the size of the BLOB. Parameters • [in] `handle` – blob handle. • [out] `size` – pointer to the BLOB size (in bytes). Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Adapting the scheduler to a hardware platform that uses processors of varying performance (sched_api.h)

The API is defined in the header file `sysroot-*-kos/include/coresrv/sched/sched_api.h` from the KasperskyOS SDK.

The API lets you set the performance levels of processors (computing cores) in the scheduler to balance workload scheduling. By default, workload scheduling is conducted based on the assumption that all processors on the platform have the same performance. However, this is not the case on hardware platforms with ARM big.LITTLE technologies and Intel E/P cores, for example. In addition, workload schedule balancing makes it possible to ensure that processors with higher performance are used more intensively while the load on lower-performing processors is reduced.

Information about API functions is provided in the table below.

Using the API

To define the processor performance levels in the scheduler, call the `KnSchedSetCpuCapacity()` function. This function uses the `cpu` and `capacity` parameters to receive the processor index and the processor performance level, respectively. The processor performance level is a relative value for which the highest-performing processor on the hardware platform is used as the baseline for comparison. Specify the `SCHED_CPU_CAPACITY_MAX` value (`1024`) for the highest-performing processor. For processors with lower performance, you need to specify a fraction of this value. For example, if this percentage is 75%, the `capacity` parameter must be set to `768`. (By default, the `SCHED_CPU_CAPACITY_MAX` performance level is assigned to all processors on the hardware platform.)

Information about API functions

sched_api.h functions

Function	Information about the function
`KnSchedSetCpuCapacity()`	Purpose Defines the processor performance level in the scheduler. Parameters • [in] `cpu` – processor index. Enumeration starts with zero. The type of processor index is defined in the header file `sysroot-*-kos/include/rtl/cpuset.h` from the KasperskyOS SDK. • [in] `capacity` – processor performance level. It takes values ranging from `SCHED_CPU_CAPACITY_MIN` to `SCHED_CPU_CAPACITY_MAX`. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If the processor index exceeds the maximum, the function returns `rcInvalidArgument`. If the processor performance level is outside of the permissible range, the function returns `rcInvalidArgument`. If the hardware platform has one processor (one computing core), the function returns `rcUnimplemented`.

Using a memory pool (mempool.h)

The API is defined in the header file `sysroot-*-kos/include/kos/mempool.h` from the KasperskyOS SDK.

The API lets you use a memory pool, which is a set of pre-allocated regions of process memory of the same size (memory blocks). Memory blocks can be taken from the pool and returned to it in a manner similar to dynamic memory allocation and deallocation, but faster and in a fixed amount of time, as no system calls are made. Memory allocation can optionally be performed when the pool is empty. However, attempts to return memory blocks to a completely full pool result in the deallocation of these blocks.

Information about API functions is provided in the table below.

Creating a pool

To create a pool, call the `KosMempoolCreate()` or `KosMempoolCreateEx()` function. These functions accept the number and size of memory blocks via the `blockCount` and `blockSize` parameters, respectively.

The `KosMempoolCreateEx()` function has additional parameters.

In the `flags` parameter, you need to specify the flags that define the parameters for receiving a memory block from the pool when it is empty. To do so, pass a structure containing the following fields to the function:

- `usePreallocatedOnly` – specifies whether to allocate a memory block (`rtl_false` – yes, `rtl_true` – no).
- `allowWaiting` – specifies whether to wait for the memory block to return to the pool or for the memory block to be freed (`rtl_true` – yes, `rtl_false` – no).

If the pool is empty, the following options are available for receiving a memory block from the pool:

- If the `usePreallocatedOnly` and `allowWaiting` parameters are set to `rtl_false`, an attempt is made to allocate a memory block. If this fails, the operation ends with an error.
- If the `usePreallocatedOnly` and `allowWaiting` parameters are set to `rtl_true`, the operation waits for the memory block to be returned to the pool, and the thread that performs the operation is blocked.
- If the `usePreallocatedOnly` parameter is set to `rtl_false` and the `allowWaiting` parameter is set to `rtl_true`, an attempt is made to allocate a memory block. If this fails, the operation waits for either the return of the memory block to the pool or the deallocation of the memory block, and the thread that performs the operation is blocked.
- If the `usePreallocatedOnly` parameter is set to `rtl_true` and the `allowWaiting` parameter is set to `rtl_false`, the operation ends with an error.

The `allocFunc`, `freeFunc`, and `sanFunc` parameters can be used to specify the callback functions that can be called to allocate, free, and clear a memory block, respectively. Callback functions for allocating and freeing a memory block are called when a pool is created or deleted. Also, these callback functions can optionally be called when performing the operations of obtaining a memory block from the pool and returning a memory block to the pool. The callback function for clearing a memory block is called when a memory block is returned to the pool, for example, to delete the contents of this block. By default, the `KosMemAlloc()` and `KosMemFree()` functions are used to allocate and free a memory block, respectively (alloc.h API). Instead of callback functions for allocating and freeing a memory block, you can specify callback functions for creating and deleting an object, for example.

The callback functions for allocating, freeing, and clearing a memory block have the input parameter `ctx`, which they use to receive a pointer to arbitrary data passed to the `KosMempoolCreateEx()` function via the parameter of the same name. The callback functions for allocating and clearing a memory block take the size of a memory block in bytes (the `size` input parameter) that is specified via the `blockSize` parameter of the `KosMempoolCreate(Ex)()` functions. The callback functions for freeing and clearing a memory block receive a pointer to the memory block (`ptr` input parameter).

The callback function for allocating a memory block, if successful, returns a pointer to the memory block, otherwise it returns `RTL_NULL`. The callback functions for freeing and clearing a memory block do not return values.

Identical calls of the `KosMempoolCreate()` and `KosMempoolCreateEx()` functions:

```
KosMempoolCreate(blockCount, size);
KosMempoolCreateEx(blockCount, size, (KosMempoolFlags){0}, RTL_NULL, RTL_NULL, RTL_NULL, RTL_NULL);
```

Getting a memory block from a pool

To get a memory block from a pool, call the `KosMempoolGet()` function. If the pool is empty, the operation of getting a memory block from the pool may be performed differently depending on how it was specified when creating the pool. The calling thread may be blocked.

Returning a memory block to the pool

To return a memory block to the pool, call the `KosMempoolPut()` function. If the pool is full, the returned memory block will be freed.

Getting information about a pool

To get the size of a memory block in the pool, call the `KosMempoolGetBlockSize()` function.

To get the data that is passed to the callback functions for allocating, freeing, and clearing a memory block, call the `KosMempoolGetContext()` function.

Deleting a pool

To delete a pool, call the `KosMempoolDestroy()` function.

> Memory blocks that are not returned to the pool (including those that were allocated after the pool was created) will not be freed.

Information about API functions

mempool.h functions

Function	Information about the function
`KosMempoolCreate()`	Purpose Creates a memory pool. Parameters • [in] `blockCount` – number of memory blocks in the pool. • [in] `blockSize` – memory block size, in bytes. Returned values: If successful, the function returns the memory pool ID, otherwise it returns `RTL_NULL`.
`KosMempoolCreateEx()`	Purpose Creates a memory pool. Parameters • [in] `blockCount` – number of memory blocks in the pool. • [in] `blockSize` – memory block size, in bytes. • [in] `flags` – flags that specify the parameters for the operation of receiving a memory block from the pool when it is empty. • [in, optional] `allocFunc` – ID of the memory block allocation callback function, or `RTL_NULL` so that the default function is used to allocate a memory block. • [in, optional] `freeFunc` – ID of the callback function for freeing the memory block, or `RTL_NULL` so that the default function is used to free the memory block. • [in, optional] `sanFunc` – ID of the callback function for clearing the memory block, or `RTL_NULL` if the memory block does not need to be cleared when returned to the pool. • [in, optional] `ctx` – pointer to the data that will be passed to the callback functions for allocating, freeing, and clearing a memory block, or `RTL_NULL` if no data is required. The callback functions will receive the passed data through their own `ctx` parameter. Returned values: If successful, the function returns the memory pool ID, otherwise it returns `RTL_NULL`.
`KosMempoolGet()`	Purpose Gets a memory block from a pool. Parameters • [in] `pool` – memory pool ID. Returned values: If successful, the function returns the pointer to the memory block, otherwise it returns `RTL_NULL`.

Function	Information about the function
`KosMempoolPut()`	Purpose Returns a memory block to a pool. Parameters • [in] `pool` – memory pool ID. • [in] `ptr` – pointer to the memory block. Returned values N/A
`KosMempoolDestroy()`	Purpose Deletes a memory pool. Parameters • [in] `pool` – memory pool ID. Returned values N/A
`KosMempoolGetBlockSize()`	Purpose Gets the size of the memory block in the pool. Parameters • [in] `pool` – memory pool ID. Returned values: Size of the memory block in the pool, in bytes.
`KosMempoolGetContext()`	Purpose Receives data that is passed to callback functions for allocating, freeing, and clearing a memory block. Parameters • [in] `pool` – memory pool ID. Returned values: Pointer to data that is passed to callback functions for allocating, freeing, and clearing a memory block, or `RTL_NULL` if no data is passed.

Using data containers (valcontainer.h)

The API is defined in the header file `sysroot-*-kos/include/kos/valcontainer.h` from the KasperskyOS SDK.

The API lets you use data containers whose elements are process memory regions with the same size and alignment. A data container is created empty. When an element is added to it, a memory region is allocated to the element and the necessary data is copied to that region. The data in elements of a container can be changed. A container can contain elements with identical data. In addition to the operation for adding an element, operations for searching for an element by contents, enumerating elements, removing an element, and cleaning a container are supported.

Information about API functions and macros is provided in the table below.

Creating a container

To create a container, call the `KosValContainerCreate()` function. The `size` and `align` parameters must be used to set the size and alignment of the elements, respectively. You can use the `ops` parameter to pass the pointer of a structure containing pointers to callback functions that are called before adding or removing an element, and the pointer of a callback function for comparing the contents of elements with a search key. In addition to pointers to callback functions, this structure contains a `ctx` field with a pointer to a data object that is used by callback functions to receive input data and/or save output data. (All three callback functions have a `ctx` parameter that they use to receive a pointer to this data object.) If this data object is not required by any of the callback functions, `RTL_NULL` must be specified in the `ctx` field.

Only part of the callback functions can be specified. Also, if you do not need to process the addition and removal of an element, and the default function is suitable for comparing the contents of elements with a search key, the `RTL_NULL` value must be specified in the `ops` parameter. By default, a function identical to the `memcmp()` function from the `libc` library API is used to compare the contents of elements with a search key. In this case, the size of the element is passed via the third parameter of the comparison function.

The callback function corresponding to the `insert` pointer is called before the element is added, that is, after the memory region is allocated and data is copied to this region, but before it is included in the list of container elements. In addition to the `ctx` parameter, this callback function has a `data` parameter that it uses to receive a pointer to the data copied to the allocated memory region. This callback function can be used, for example, to check or process the contents of an added element. If the return code has the value `rcOk`, the allocated memory region is included in the list of container elements. Otherwise, it is freed.

The callback function corresponding to the `remove` pointer is called before the element is removed, that is, before the memory region is excluded from the list of container elements, after which this region is freed. In addition to the `ctx` parameter, this callback function has a `data` parameter that it uses to receive a pointer to the contents of the element being removed. This callback function can be used, for example, to free resources.

The callback function corresponding to the `compare` pointer is used to compare the contents of elements with a search key. Along with the `ctx` parameter, this callback function has the `data` and `key` parameters that it uses to receive pointers to the contents of the element and the search key, respectively. This callback function can be used, for example, to perform a search by hash sums of the contents of elements instead of the default byte-by-byte comparison. If the contents of an element match the search key, the return code must be `0`. Otherwise, it must be a non-zero value.

Adding an element to a container

To add an element to a container, call the `KosValContainerInsert()` function. You can add elements that have identical contents.

Getting a container element

To get a container element with the necessary contents, call the `KosValContainerGet()` function. If the container contains several elements with identical contents, this function will return the pointer for the element that was added first.

Removing an element from a container

To remove an element with the necessary contents from the container, call the `KosValContainerRemove()` function. If the container contains several elements with identical contents, the element that was added first will be removed.

Navigation within a container

To initialize an iterator for navigation within a container and for getting its elements, call the `KosValContainerIteratorInit()` function. After initialization, the iterator is in the initial position.

To move the iterator to the next position, call the `KosValContainerIteratorNext()` function.

To check whether the iterator is in the final position, call the `KosValContainerIteratorIsEnd()` function.

To get the container element corresponding to the current position of the iterator, call the `KosValContainerIteratorGetData()` function.

Traversing container elements

To traverse container elements, use the `KOS_VAL_CONTAINER_FOR_EACH()` macro. This macro is expanded into the header of the `for` operator. The pointer defined in the `data` parameter will contain the address of the next element in each iteration. You can use this pointer to change the contents of elements, for example.

Enumerating container elements

To enumerate elements of a container, use the `KosValContainerEnum()` function. The order in which the elements are enumerated corresponds to the order in which they were added to the container.

Getting the number of elements in a container

To get the number of elements in a container, call the `KosValContainerCount()` function.

Clearing a container

To remove all elements from a container, call the `KosValContainerClear()` function.

Deleting a container

To delete a container, call the `KosValContainerDestroy()` function. When you delete a container, all of its elements are deleted.

Information about API functions and macros

valcontainer.h functions and macros

Function/Macro	Information about the function/macro
`KosValContainerCreate()`	Purpose Creates a container. Parameters • [in] `size` – size (in bytes) of the container elements. • [in] `align` – value defining the container elements alignment. It must be a power of two. The addresses of container elements may be unaligned (*align=1*) or aligned (*align=2,4,...,2^N*) to the boundary of a *2^N*-byte sequence (for example, two-byte or four-byte). • [in,optional] `ops` – pointer to the structure that contains pointers to callback functions that are called before adding or removing an element, and the pointer to a callback function for comparing the contents of elements with a search key. This structure also contains a `ctx` field with a pointer to a data object that is used by callback functions to receive input data and/or save output data. The parameter must be set to `RTL_NULL` if you do not need to process the addition and removal of an element, but instead the default function is sufficient to compare the contents of elements with a search key. • [out] `outContainer` – pointer to the container ID. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosValContainerDestroy()`	Purpose Deletes a container. Parameters • [in] `container` – ID of the container. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function/Macro	Information about the function/macro
`KosValContainerInsert()`	Purpose Adds an element to a container. Parameters • [in] `container` – ID of the container. • [in] `data` – pointer to the data to be copied to the added container element. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosValContainerRemove()`	Purpose Removes the element whose contents match the specified search key from the container. Parameters • [in] `container` – ID of the container. • [in] `key` – pointer to the search key. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosValContainerEnum()`	Purpose Enumerates the elements of a container. Parameters • [in] `container` – ID of the container. • [in] `index` – index for enumerating container elements. Enumeration starts with zero. Returned values: Pointer to the element of the container, or `RTL_NULL` if the index value is exceeded.
`KosValContainerClear()`	Purpose Clears a container. Parameters • [in] `container` – ID of the container. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosValContainerGet()`	Purpose Gets the container element whose contents match the specified search key. Parameters • [in] `container` – ID of the container. • [in] `key` – pointer to the search key. Returned values: Pointer to the container element, or `RTL_NULL` if a container element whose contents match the specified search key was not found.

Function/Macro	Information about the function/macro
`KosValContainerCount()`	Purpose Gets the number of elements in a container. Parameters • [in] `container` – ID of the container. Returned values: Number of elements in a container.
`KosValContainerIteratorInit()`	Purpose Initializes an iterator for navigation within a container and for getting its elements. Parameters • [in] `container` – ID of the container. Returned values: Initialized iterator for navigation within the container and for getting its elements.
`KosValContainerIteratorIsEnd()`	Purpose Checks whether the iterator is in the final position. Parameters • [in] `iterator` – pointer to the iterator. Returned values: If the verification is successful, the function returns `rtl_true`, otherwise it returns `rtl_false`. The type of returned value is defined in the header file `sysroot-*-kos/include/rtl/stdbool.h` from the KasperskyOS SDK.
`KosValContainerIteratorGetData()`	Purpose Gets the element of the container corresponding to the current position of the iterator. Parameters • [in] `iterator` – pointer to the iterator. Returned values: Pointer to the element of the container, or `RTL_NULL` if the iterator is at the end position.
`KosValContainerIteratorNext()`	Purpose Moves the iterator to the next position. Parameters • [in] `iterator` – pointer to the iterator. Returned values N/A

Function/Macro	Information about the function/macro
`KOS_VAL_CONTAINER_FOR_EACH()`	Purpose Traverses elements in a container. Parameters • [in] `container` – ID of the container. • [in] `data` – pointer for saving the addresses of container elements when traversing them. Macro values N/A

POSIX support

POSIX support limitations

KasperskyOS has a limited implementation of POSIX oriented toward the POSIX.1-2008 standard. These limitations are primarily due to security precautions.

> There is no XSI support or optional functionality.

Limitations affect the following:

- Interaction between processes
- Interaction between threads via signals
- Asynchronous input/output
- Use of robust mutexes
- Terminal operations
- Shell operations
- File handle management
- Clock usage
- Getting system parameters

Limitations include:

- Unimplemented interfaces
- Interfaces that are implemented with deviations from the POSIX.1-2008 standard
- Stub interfaces that do not perform any operations except assign the `ENOSYS` value to the `errno` variable and return the value `-1`

> In KasperskyOS, signals cannot interrupt the `Call()`, `Recv()`, and `Reply()` system calls that support the operation of libraries that implement the POSIX interface.
> The KasperskyOS kernel does not transmit signals.
> On hardware platforms running an Arm processor architecture, input and output parameters of POSIX interfaces cannot be saved in "Device memory" because this could lead to unpredictable behavior. Parameters of POSIX interfaces should be saved in "Normal memory". To copy data from "Device memory" to "Normal memory" and vice versa, you must use the `RtlPedanticMemcpy()` function declared in the header file `sysroot-*-kos/include/rtl/string_pedantic.h` from the KasperskyOS SDK.

Limitations on interaction between processes

Interface	Purpose	Implementation	Header file based on the POSIX.1-2008 standard
`fork()`	Create a new (child) process.	Not implemented	`unistd.h`
`pthread_atfork()`	Register the handlers that are called before and after the child process is created.	Not implemented	`pthread.h`
`wait()`	Wait for the child process to stop or complete.	Stub	`sys/wait.h`
`waitid()`	Wait for the state of the child process to change.	Not implemented	`sys/wait.h`
`waitpid()`	Wait for the child process to stop or complete.	Stub	`sys/wait.h`
`execl()`	Run the executable file.	Not implemented	`unistd.h`
`execle()`	Run the executable file.	Not implemented	`unistd.h`
`execlp()`	Run the executable file.	Not implemented	`unistd.h`
`execv()`	Run the executable file.	Not implemented	`unistd.h`
`execve()`	Run the executable file.	Not implemented	`unistd.h`

Interface	Purpose	Implementation	Header file based on the POSIX.1-2008 standard
`execvp()`	Run the executable file.	Not implemented	`unistd.h`
`fexecve()`	Run the executable file.	Not implemented	`unistd.h`
`setpgid()`	Move the process to another group or create a group.	Stub	`unistd.h`
`setsid()`	Create a session.	Stub	`unistd.h`
`getpgrp()`	Get the group ID of the calling process.	Stub	`unistd.h`
`getpgid()`	Get the group ID.	Stub	`unistd.h`
`getppid()`	Get the ID of the parent process.	Stub	`unistd.h`
`getsid()`	Get the session ID.	Stub	`unistd.h`
`times()`	Get the time values for the process and its descendants.	Stub	`sys/times.h`
`kill()`	Send a signal to the process or group of processes.	Only the `SIGTERM` signal can be sent. The `pid` parameter is ignored.	`signal.h`
`pause()`	Wait for a signal.	Stub	`unistd.h`
`sigpending()`	Check for received blocked signals.	Not implemented	`signal.h`
`sigqueue()`	Send a signal to the process.	Not implemented	`signal.h`
`sigtimedwait()`	Wait for a signal from the defined set of signals.	Not implemented	`signal.h`
`sigwaitinfo()`	Wait for a signal from the defined set of signals.	Not implemented	`signal.h`
`sem_init()`	Create an unnamed semaphore.	You cannot create an unnamed semaphore for synchronization between processes. If a non-zero value is passed through the `pshared` parameter, it will return only the value `-1` and will assign the `ENOTSUP` value to the `errno` variable.	`semaphore.h`
`sem_open()`	Create/open a named semaphore.	You cannot open a named semaphore that was created by another process. Named semaphores (like unnamed semaphores) are local, which means that they are accessible only to the process that created them.	`semaphore.h`
`pthread_spin_init()`	Create a spin lock.	You cannot create a spin lock for synchronization between processes. If the `PTHREAD_PROCESS_SHARED` value is passed through the `pshared` parameter, this value will be ignored.	`pthread.h`
`mmap()`	Map to memory.	You cannot perform memory mapping for interaction between processes. If the `MAP_SHARED` and `PROT_WRITE` values are passed through the `flags` and `prot` parameters, respectively, it will return only the `MAP_FAILED` value and will assign the `EACCES` value to the `errno` variable. For all other possible values of the `prot` parameter, the `MAP_SHARED` value of the `flags` parameter is ignored. In addition, you cannot use the `prot` parameter to pass combinations of the `PROT_WRITE|PROT_EXEC` and `PROT_READ|PROT_WRITE|PROT_EXEC` flags, or combinations with the `PROT_READ|PROT_EXEC` flag under certain configurations of the KasperskyOS kernel. In this case, it will return only the `MAP_FAILED` value and will assign the `ENOMEM` value to the `errno` variable.	`sys/mman.h`

Interface	Purpose	Implementation	Header file based on the POSIX.1-2008 standard
`mprotect()`	Define the memory access permissions.	For security purposes, some configurations of the KasperskyOS kernel prohibit granting simultaneous write-and-execute access to virtual memory regions. If this type of kernel configuration is in use and you pass the `PROT_WRITE|PROT_EXEC` value through the `prot` parameter, it will only return the `-1` value and will assign the `ENOTSUP` value to the `errno` variable.	`sys/mman.h`
`pipe()`	Create an unnamed channel.	You cannot use an unnamed channel for data transfer between processes. Unnamed channels are local, which means that they are accessible only to the process that created them.	`unistd.h`
`mkfifo()`	Create a special FIFO file (named channel).	Stub	`sys/stat.h`

Limitations on interaction between threads via signals

Interface	Purpose	Implementation	Header file based on the POSIX.1-2008 standard
`pthread_kill()`	Send a signal to a thread.	You cannot send a signal to a thread. If a signal number is passed through the `sig` parameter, only the `ENOSYS` value is returned.	`signal.h`
`siglongjmp()`	Restore the state of the control thread and the signals mask.	Not implemented	`setjmp.h`
`sigsetjmp()`	Save the state of the control thread and the signals mask.	Not implemented	`setjmp.h`

Asynchronous input/output limitations

Interface	Purpose	Implementation	Header file based on the POSIX.1-2008 standard
`aio_cancel()`	Cancel input/output requests that are waiting to be handled.	Not implemented	`aio.h`
`aio_error()`	Receive an error from an asynchronous input/output operation.	Not implemented	`aio.h`
`aio_fsync()`	Request the execution of input/output operations.	Not implemented	`aio.h`
`aio_read()`	Request a file read operation.	Not implemented	`aio.h`
`aio_return()`	Get the status of an asynchronous input/output operation.	Not implemented	`aio.h`
`aio_suspend()`	Wait for the completion of asynchronous input/output operations.	Not implemented	`aio.h`
`aio_write()`	Request a file write operation.	Not implemented	`aio.h`
`lio_listio()`	Request execution of a set of input/output operations.	Not implemented	`aio.h`

Limitations on the use of robust mutexes

Interface	Purpose	Implementation	Header file based on the POSIX.1-2008 standard
`pthread_mutex_consistent()`	Return a robust mutex to a consistent state.	Not implemented	`pthread.h`
`pthread_mutexattr_getrobust()`	Get a robust mutex attribute.	Not implemented	`pthread.h`
`pthread_mutexattr_setrobust()`	Define a robust mutex attribute.	Not implemented	`pthread.h`

Terminal operation limitations

Interface	Purpose	Implementation	Header file based on the POSIX.1-2008 standard
`ctermid()`	Get the path to the file of the control terminal.	Only returns or passes an empty string through the `s` parameter.	`stdio.h`
`tcsetattr()`	Define the terminal settings.	The input speed, output speed, and other settings specific to hardware terminals are ignored.	`termios.h`
`tcdrain()`	Wait for output completion.	Returns only the value `-1`.	`termios.h`
`tcflow()`	Suspend or resume receipt or transmission of data.	Suspending output and resuming suspended output are not supported.	`termios.h`
`tcflush()`	Clear the input queue or output queue, or both of these queues.	Returns only the value `-1`.	`termios.h`
`tcsendbreak()`	Break the connection with the terminal for a set time.	Returns only the value `-1`.	`termios.h`
`ttyname()`	Get the path to the terminal file.	Returns only a null pointer.	`unistd.h`
`ttyname_r()`	Get the path to the terminal file.	Returns only an error value.	`unistd.h`
`tcgetpgrp()`	Get the ID of a group of processes using the terminal.	Returns only the value `-1`.	`unistd.h`
`tcsetpgrp()`	Define the ID for a group of processes using the terminal.	Returns only the value `-1`.	`unistd.h`
`tcgetsid()`	Get the ID of a group of processes for the leader of the session connected to the terminal.	Returns only the value `-1`.	`termios.h`

Shell operation limitations

Interface	Purpose	Implementation	Header file based on the POSIX.1-2008 standard
`popen()`	Create a child process for command execution and an unnamed channel with this process.	Not implemented	`stdio.h`
`pclose()`	Close the unnamed channel with the child process created by `popen()`, and wait for the child process to terminate.	Not implemented	`stdio.h`
`system()`	Create a child process for command execution.	Stub	`stdlib.h`

Interface	Purpose	Implementation	Header file based on the POSIX.1-2008 standard
`wordexp()`	Perform a shell-like expansion of the string.	Not implemented	`wordexp.h`
`wordfree()`	Free up the memory allocated for the results from calling `wordexp()`.	Not implemented	`wordexp.h`

Limitations on file handle management

Interface	Purpose	Implementation	Header file based on the POSIX.1-2008 standard
`dup()`	Make a copy of the handle of an opened file.	Handles of regular files, standard I/O streams, sockets and channels are supported. There is no guarantee that the lowest available handle will be received.	`unistd.h`
`dup2()`	Make a copy of the handle of an opened file.	Handles of regular files, standard I/O streams, sockets and channels are supported. The handle of an opened file needs to be passed through the `fildes2` parameter.	`unistd.h`

Limitations on clock usage

Interface	Purpose	Implementation	Header file based on the POSIX.1-2008 standard
`clock_gettime()`	Get the time value.	If the `CLOCK_PROCESS_CPUTIME_ID` value or `CLOCK_THREAD_CPUTIME_ID` value is passed through the `clock_id` parameter, it will return only the value `-1` and will assign the `EINVAL` value to the `errno` variable.	`time.h`
`clock()`	Get the CPU time spent on execution of the calling process.	Returns the amount of time (in milliseconds) that has elapsed since the KasperskyOS kernel was started.	`time.h`

Getting system parameters

Interface	Purpose	Implementation	Header file based on the POSIX.1-2008 standard
`confstr()`	Get a system parameter.	Stub	`unistd.h`

POSIX implementation specifics

In KasperskyOS, the specific implementation of some POSIX interfaces not entirely defined by the POSIX.1-2008 standard differs from the implementation of these interfaces in Linux and other UNIX-like operating systems. Information about these interfaces is provided in the table below.

Interface	Purpose	Implementation	Header file based on the POSIX.1-2008 standard
`bind()`	Assign a name to a socket.	When using a VFS version that supports only network operations, files of sockets in the `AF_UNIX` family are saved in a special file system implemented by this VFS version when `bind()` is called. A socket file can be created only in the root of the file system or in the `/tmp` directory, and it can be re-used after the socket is closed.	`sys/socket.h`
`mmap()`	Map to memory.	The `MAP_FIXED` flag in the `flags` parameter is not supported. Mapping more than 4 GB is not supported on hardware platforms running an AArch64 (Arm64) processor architecture.	`sys/mman.h`
`read()`	Read from a file.	If the size of the `buf` buffer exceeds the size of the read data, the remainder of this buffer is filled with zeros.	`unistd.h`
`poll()`	Track the state of file handles.	You cannot combine invalid file handles (for example, closed handles, unavailable handles, or handles with invalid values) with a negative `timeout` parameter value (unlimited timeout). In this case, the function returns `-1`, and `errno` is set to `EINVAL`.	`poll.h`
`select()`	Track the state of file handles.	You cannot call a function with null values for all parameters. In this case, the function returns `-1`, and `errno` is set to `EINVAL`.	`sys/select.h`
`fcntl()`	Perform operations with a file handle.	The Linux-specific `F_SETPIPE_SZ` and `F_GETPIPE_SZ` macros are supported.	`fcntl.h`

In KasperskyOS, the `errno` variable can take the values of both standard POSIX error codes and those specific to KasperskyOS. For information about error codes specific to KasperskyOS, see Return codes.

Concurrently using POSIX and the libkos API

In a thread that is created using Pthreads, you cannot use the following `libkos` APIs:

- event.h, mutex.h, rwlock.h, semaphore.h, condvar.h
- thread.h, thread_api.h
- dma.h
- mmio.h
- irq.h

The following `libkos` APIs can be used together with Pthreads (and other POSIX APIs):

- handle_api.h
- notice_api.h
- task.h, task_api.h
- queue.h

POSIX interfaces cannot be used in threads that were created using the thread.h and thread_api.h APIs.

The syscalls.h API can be used in any threads that were created using Pthreads or the thread.h and thread_api.h APIs.

Obtaining statistical data on the system

The `libkos` and `libc` libraries provide APIs for obtaining statistical data on the system. This data includes the following information:

- CPU time usage by the system and by an individual process
- Memory usage by the system and by an individual process
- Info on processes and threads
- Info on file systems and network interfaces

Obtaining statistical data on the system through the libkos library API

The `libkos` library provides an API that obtains statistical data on CPU time and memory usage, and info on processes and threads. This API is defined in the header file `sysroot-*-kos/include/coresrv/stat/stat_api.h` from the KasperskyOS SDK.

> The API includes `KnGroupStatGetParam()`, `KnCpuStatGetParam()`, `KnTaskStatGetParam()` functions that request the values of performance counters. To use these functions, you need to build a solution with a KasperskyOS kernel version that supports the full set of performance counters (for details, see image library). An exception to this rule includes calls of the `KnGroupStatGetParam()` function with the `GROUP_PARAM_MEM_FREE` and `GROUP_PARAM_MEM_TOTAL` values in the `param` parameter (see the table containing information about memory usage below) that can be successfully executed without a special kernel version.

Receiving information about CPU time

Uptime of CPUs (processor cores) is counted from the startup of the KasperskyOS kernel.

To obtain information about CPU time usage, use the `KnGroupStatGetParam()`, `KnTaskStatGetParam()` and `KnCpuStatGetParam()` functions. The values provided in the table below need to be passed in the `param` parameter of these functions. The CPU time obtained by calling the `KnGroupStatGetParam()`, `KnTaskStatGetParam()` or `KnCpuStatGetParam()` function is presented in nanoseconds. The CPU index (enumeration starts with zero) is the input parameter of the `KnCpuStatGetParam()` function. To get the total number of processors (computing cores) on the hardware platform, call the `KnHalGetCpuCount()` function from the hal_env.h API.

Information about CPU time

Function	Value of the param parameter	Obtained value
KnGroupStatGetParam()	GROUP_PARAM_CPU_KERNEL	Total uptime of all processors in kernel mode
KnGroupStatGetParam()	GROUP_PARAM_CPU_USER	Total uptime of all processors in user mode
KnGroupStatGetParam()	GROUP_PARAM_CPU_IDLE	Total uptime of all processors in idle mode
KnTaskStatGetParam()	TASK_PARAM_TIME_TOTAL	Total uptime of all processors used for execution of the defined process
KnTaskStatGetParam()	TASK_PARAM_TIME_USER	Total uptime of all processors used for execution of the defined process in user mode
KnCpuStatGetParam()	CPU_STAT_PARAM_IDLE	Uptime of the defined processor in idle mode
KnCpuStatGetParam()	CPU_STAT_PARAM_USER	Uptime of the defined processor in user mode
KnCpuStatGetParam()	CPU_STAT_PARAM_KERNEL	Uptime of the defined processor in kernel mode

Receiving information about memory usage

To receive information about memory usage, you need to use the `KnGroupStatGetParam()` and `KnTaskStatGetParam()` functions. The values provided in the table below need to be passed in the `param` parameter of these functions. The memory size obtained by calling the `KnGroupStatGetParam()` or `KnTaskStatGetParam()` function is presented as the number of memory pages. The size of a memory page is 4 KB for all hardware platforms supported by KasperskyOS.

> The size of physical memory used by a process refers only to the memory allocated directly for this process. For example, if the memory of a process is mapped to an MDL buffer created by another process, the size of this buffer is not included in this value.

Information about memory usage

Function	Value of the param parameter	Obtained value
KnGroupStatGetParam()	GROUP_PARAM_MEM_TOTAL	Size of all installed physical memory.
KnGroupStatGetParam()	GROUP_PARAM_MEM_FREE	Size of free physical memory.
KnGroupStatGetParam()	GROUP_PARAM_MEM_EXEC	Size of physical memory with the "execution access" attribute.
KnGroupStatGetParam()	GROUP_PARAM_MEM_SHARED	Size of physical memory used as shared memory.
KnGroupStatGetParam()	GROUP_PARAM_MEM_DEDUPLICATED	Size of physical memory freed by zero pages deduplication allocated for committing virtual memory in "lazy" mode. (Calculated from the moment the KasperskyOS kernel is started.)
KnGroupStatGetParam()	GROUP_PARAM_MEM_DEDUP_QUEUE_LEN	Size of physical memory that can be freed on the next cycle of zero pages deduplication allocated for committing virtual memory in "lazy" mode. (This is the number of physical memory pages allocated for committing virtual memory in "lazy" mode during the time elapsed after the previous deduplication cycle until the `KnGroupStatGetParam()` function is called with the `param` parameter set to `GROUP_PARAM_MEM_DEDUP_QUEUE_LEN`. Only those pages of physical memory that remain zero will be freed before the start of the next deduplication cycle.)
KnGroupStatGetParam()	GROUP_PARAM_MEM_DEDUP_IS_ENABLED	This value indicates whether zero page deduplication allocated for committing virtual memory in "lazy" mode is enabled (`1` – enabled, `0` – disabled; it is disabled by the kernel startup parameter `vmm.zero_dedup=0`).
KnTaskStatGetParam()	TASK_PARAM_MEM_PHY	Size of physical memory used by the defined process.

Function	Value of the param parameter	Obtained value
KnTaskStatGetParam()	TASK_PARAM_MEM_VIRT	Size of virtual memory of the defined process.
KnTaskStatGetParam()	TASK_PARAM_MEM_SHARED	Size of virtual memory of the defined process mapped to shared physical memory.
KnTaskStatGetParam()	TASK_PARAM_MEM_PAGE_TABLE	Size of the page table of the defined process.
KnTaskStatGetParam()	TASK_PARAM_VMM_FRAG_INDEX	Average size of free virtual memory regions of the defined process.
KnTaskStatGetParam()	TASK_PARAM_VMM_MAX_CONTIGIOUS_SPACE	Maximum size of free virtual memory regions of the defined process.

Obtaining information on processes and threads

In addition to information about CPU time and memory usage, the `KnGroupStatGetParam()` and `KnTaskStatGetParam()` functions also let you obtain information on processes and threads. To receive this information, the values provided in the table below need to be passed through the `param` parameter of these functions.

Information on processes and threads

Function	Value of the param parameter	Obtained value
KnGroupStatGetParam()	GROUP_PARAM_TASKS	Number of processes (excluding the kernel process)
KnGroupStatGetParam()	GROUP_PARAM_THREADS	Number of threads (including kernel threads)
KnTaskStatGetParam()	TASK_PARAM_PPID	ID of the parent process of the defined process (PPID)
KnTaskStatGetParam()	TASK_PARAM_PRIO	Priority of the initial thread of the defined process
KnTaskStatGetParam()	TASK_PARAM_STATE	State of the defined process (according to the list of `TaskExecutionStates` defined in the header file `sysroot-*-kos/include/task/pcbpage.h` from the KasperskyOS SDK)
KnTaskStatGetParam()	TASK_PARAM_IMGSIZE	Image size of the program loaded into the memory of the defined process (in bytes)
KnTaskStatGetParam()	TASK_PARAM_TIME_START	Time (in nanoseconds) between startup of the kernel and startup of the defined process
KnTaskStatGetParam()	TASK_PARAM_HANDLES	Number of handles owned by the defined process
KnTaskStatGetParam()	TASK_PARAM_THREADS	Number of threads in the defined process

In addition to the `KnGroupStatGetParam()` and `KnTaskStatGetParam()` functions, information on processes and threads can also be obtained by using the following functions:

- `KnTaskStatGetName()` – gets the name of the defined process.
- `KnTaskStatGetPath()` – gets the name of the executable file in ROMFS that was used to start the defined process.

  This function can be used only if the process was started from an executable file in ROMFS. Otherwise, the function call will result in an empty string.
- `KnTaskStatGetId()` – gets the ID of the defined process (PID).
- `KnTaskGetIdByName()` gets the ID of the process (PID) with the defined name.

Obtaining information on CPU time and memory usage by each process

To get information about CPU time and memory usage by each process, do the following:

1. Get the list of processes by calling the `KnGroupStatGetTaskList()` function.
2. Get the number of items on the list of processes by calling the `KnTaskStatGetTasksCount()` function.
3. Iterate through the list of processes, repeating the following steps:
   a. Get an item from the list of processes by calling the `KnTaskStatEnumTaskList()` function.
   b. Get the process name by calling the `KnTaskStatGetName()` function.
      This is necessary to identify the process for which the information about CPU time and memory usage will be received.
   c. Get information about CPU time and memory usage by calling the `KnTaskStatGetParam()` function.
   d. Verify that the process was not terminated. If the process has terminated, discard the obtained information about the CPU time and memory used by this process.
      To verify that the process was not terminated, you need to call the `KnTaskStatGetParam()` function, using the `param` parameter to pass the `TASK_PARAM_STATE` value. A value other than `TaskStateTerminated` should be received.
   e. Finish working with the item on the list of processes by calling the `KnTaskStatCloseTask()` function.
4. Finish working with the list of processes by calling the `KnTaskStatCloseTaskList()` function.

Calculating CPU load

Indicators of load on CPUs (processor cores) may be the following values:

- Percent load of all processors
- Percent load of all processors by each process
- Percent load of each processor

These indicators are calculated for a specific time interval, at the start and end of which the information about CPU time utilization was received. (For example, CPU load can be monitored with periodic receipt of information about CPU time utilization.) The values obtained at the start of the interval need to be subtracted from the values obtained at the end of the interval. In other words, the following increments need to be obtained for the interval:

- *TK* – uptime of all processors in kernel mode.
- *TKi [i=1,2,...,n]* – uptime of the *i*th processor in kernel mode.
- *TU* – uptime of all processors in user mode.
- *TUi [i=1,2,...,n]* – uptime of the *i*th processor in user mode.

- *TIDLE* – uptime of all processors in idle mode.
- *TIDLEi [i=1,2,...,n]* – uptime of the *i*th processor in idle mode.
- *Tj [j=1,2,...,m]* – CPU time spent on execution of the *j*th process.

The percent load of all processors is calculated as follows:

(TK+TU)/(TK+TU+TIDLE).

The percent load of the *i*th processor is calculated as follows:

(TKi+TUi)/(TKi+TUi+TIDLEi).

The percent load of all processors caused by the *j*th process is calculated as follows:

Tj/(TK+TU+TIDLE).

Obtaining statistical data on the system through the libc library API

The `libkos` library provides APIs that let you obtain statistical data on file systems and network interfaces managed by VFS. The functions of these APIs are presented in the table below.

Information on file systems and network interfaces

Function	Header file from the KasperskyOS SDK	Obtained information
`statvfs()`	`sysroot-*-kos/include/strict/posix/sys/statvfs.h`	File system information, such as the block size, number of blocks, and number of available blocks
`getvfsstat()`	`sysroot-*-kos/include/sys/statvfs.h`	The information on all mounted file systems is identical to the information provided by the `statvfs()` function
`getifaddrs()`	`sysroot-*-kos/include/ifaddrs.h`	Information on network interfaces, such as their name, IP address, and subnet mask

Dynamically creating IPC channels using the DCM system program

The KasperskyOS SDK provides the `DCM` (Dynamic Connection Manager) system program, which lets you dynamically create IPC channels. This program enables servers to notify clients about their provided endpoints, and pass callable handles for using these endpoints.

To use the `DCM` program, the KasperskyOS SDK provides the following files:

- `sysroot-*-kos/include/coresrv/dcm/dcm_api.h` – header file that defines the basic API
- `sysroot-*-kos/include/coresrv/dcm/helpers/connect.h` – header file that defines the auxiliary API for clients
- `sysroot-*-kos/include/coresrv/dcm/groups/pub.h` – header file that defines the additional API for servers

- `sysroot-*-kos/include/coresrv/dcm/groups/subscr.h` – header file that defines the additional API for clients
- `sysroot-*-kos/include/coresrv/dcm/dcm.h` – header file for including clients and servers into source code
- `sysroot-*-kos/bin/dcm` – executable file
- `sysroot-*-kos/include/kl/core/DCM.*dl` – formal specification files

The `DCM` program is included in the KasperskyOS boot image and is automatically added to each solution image.

IPC channels from clients and servers to the `DCM` must be statically created. The `libkos` library serves as the client library of the `DCM` program.

You can run multiple `DCM` processes. Each client and server can be connected to only one `DCM` process.

The `DCM` API uses IDs known as *DCM handles* . DCM handles identify objects that provide the following capabilities to clients and servers that use `DCM`:

- Clients can receive notifications about the publishing and unpublishing of endpoints, and can make requests to create IPC channels to servers.
- Servers can receive client requests to create IPC channels, and can accept or decline these requests.

The names of components, endpoints, and endpoint interfaces should be defined according to the formal specifications of solution components. (For information about the qualified name of an endpoint and component instance, see Binding methods of security models to security events.) Instead of the qualified name of an endpoint or component instance, you can use any conditional name. The names of clients and servers are defined in the init description.

Using the base API (dcm_api.h)

The API is defined in the header file `sysroot-*-kos/include/coresrv/dcm/dcm_api.h` from the KasperskyOS SDK.

General information about using the API:

- Servers publish endpoints and/or component instances (endpoint containers) to `DCM`.
- Clients receive notifications regarding the publication of individual endpoints and/or component instances.
- A separate IPC channel is created to enable use of each endpoint of a server. This IPC channel is identified on the client side by a callable handle.
- If a client needs to use a server endpoint, this client can make a request to the server to create one IPC channel to use that endpoint.
- If a client needs to use a set of server endpoints provided by a component instance, this client can make a request to the server to create multiple IPC channels to use that set of endpoints.

Information about API functions is provided in the table below.

Dynamically creating IPC channels on the server side

Dynamic creation of IPC channels on the server side includes the following steps:

1. Create a connect to `DCM` by calling the `DcmInit()` function.

2. Publish the provided endpoints and/or instances of components to `DCM` by using the `DcmPublishEndpoint()` and/or `DcmPublishComponent()` function.

   The `DcmPublishEndpoint()` function publishes the server endpoint, and the `DcmPublishComponent()` function publishes the server component instance.

   In the `serverHandle` parameter of the `DcmPublishEndpoint()` and `DcmPublishComponent()` functions, you need to specify the handle that identifies the server. The server-identifying handle can be the handle of this server process that was obtained by sequentially calling the `KosTaskGetSelf()` and `KosTaskGetHandle()` functions from the task.h API, or the handle that was created by calling the `KnHandleCreateUserObject()` function from the handle_api.h API. (When calling the `KnHandleCreateUserObject()` function, the `rights` parameter must specify the `OCAP_HANDLE_TRANSFER` and `OCAP_HANDLE_GET_SID` flags.) Clients can receive the descendant of the handle that identifies the server process via the optional `outServerHandle` parameter of the `DcmReadPubQueue()` function. After the endpoint or component instance has been published, the server-identifying handle can be closed but only if this handle was not obtained by calling the `KosTaskGetHandle()` function. (A handle obtained by calling the `KosTaskGetHandle()` function cannot be closed because the closed handle could then be used to identify other resources, which will lead to errors.)

   A `DcmPublishEndpoint()` function call creates a DCM handle that enables receipt of client requests to create an IPC channel to use a specific endpoint. A `DcmPublishComponent()` function call creates a DCM handle that enables receipt of client requests to create multiple IPC channels to use the set of endpoints of the defined component instance.

   To unpublish an endpoint or component instance, call the `DcmCloseHandle()` function to close the DCM handle that was created when they published. If a server terminates, all of its published endpoints and instances of server components will also be unpublished.

3. Receive a client request to create an IPC channel (or channels) by calling the `DcmListen()` function.

   Calls of the `DcmListen()` function with the same DCM handle in the `pubHandle` parameter let you receive only those client requests that are aimed at creating one or more IPC channels to use the endpoint or the set of endpoints of the component instance that were specified when creating this DCM handle at step 2. (The server also receives client requests that were made before the endpoint or component instance was published.)

   You can use the optional `outClientHandle` and `outClientName` parameters of the `DcmListen()` function to get the handle identifying the client and the client name, respectively. (The handle identifying the client is the descendant of the handle specified in the `clientHandle` parameter of the `DcmCreateConnection()` function called on the client side.) This data can be used to make a decision on whether to accept or reject a client request. For example, the handle identifying the client may be used to get information about the client from other processes or to get the SID for this handle to compare it to other client-identifying handles available to the server. After making a decision, you can close the handle that identifies the client.

   A `DcmListen()` function call creates a DCM handle that enables acceptance or rejection of a client request to create an IPC channel (or channels).

4. Accept a client request to create one or more IPC channels by calling the `DcmAcceptEndpointConnection()` or `DcmAcceptComponentConnection()` function.

   The `DcmAcceptEndpointConnection()` function accepts a client request to create an IPC channel to use a server endpoint, and the `DcmAcceptComponentConnection()` function accepts a client request to create IPC channels to use the set of endpoints of a server component instance.

In the `connectionId` parameter of the `DcmAcceptEndpointConnection()` and `DcmAcceptComponentConnection()` functions, specify the DCM handle that was obtained at step 3.

In the `sessionHandleDesc` parameter of the `DcmAcceptEndpointConnection()` function, you need to specify the handle transport container that contains the callable handle for using the endpoint. In the `sessionHandlesCnt` parameter of the `DcmAcceptComponentConnection()` function, you need to specify the number of endpoints provided by the component instance, which is the number of endpoints listed in the CDL file. Use the `sessionHandleDescs` parameter of the `DcmAcceptComponentConnection()` function to define an array of handle transport containers that contain callable handles for using the set of endpoints of the component instance. The order of elements in the array must match the order in which the endpoints are listed in the CDL file. You can grant access to only a portion of the endpoints of a component instance. For endpoints that will not be provided to the client, the `INVALID_HANDLE` value must be specified instead of callable handles. At least one element of the array must contain a callable handle.

To create a callable handle, call the `KnHandleCreateUserObjectEx()` function from the handle_api.h API. When calling the `KnHandleCreateUserObjectEx()` function, the `rights` parameter must specify the `OCAP_IPC_CALL` and `OCAP_HANDLE_TRANSFER` flags, and the `ipcChannel` and `riid` parameters must specify the server IPC handle of the created IPC channel and the endpoint ID (RIID), respectively. The server IPC handle is used to initialize IPC transport and manage IPC request processing on the server side. You can use the `KnHandleCreateListener()` function from the handle_api.h API to create a server handle. The endpoint ID (RIID) is a constant in the automatically generated transport code, for example: `FsDriver_operationsComp_fileOperations_iid`. After the client request is accepted, the callable handle can be closed.

To place a callable handle into the handle transport container, use the `nk_handle_desc()` macro that is defined in the header file `sysroot-*-kos/include/nk/types.h` from the KasperskyOS SDK. For the second parameter of the `nk_handle_desc()` macro, you can specify the `NK_SAME` macro that is defined in the same header file. In this case, the descendant of the transferred callable handle will have the same permissions mask as the transferred callable handle. The transfer of a callable handle to a client can be associated with a resource transfer context object by specifying the handle of this object in the third parameter of the `nk_handle_desc()` macro. This will let you use the notice_api.h API to track the closing or revocation of descendants of the callable handle that form the handle inheritance subtree whose root node was generated when this callable handle was transferred to the client. The transferred callable handle can be closed by the client, but this handle is also closed when the client terminates. To create a resource transfer context object, call the `KnHandleCreateBadge()` function from the handle_api.h API.

You can use the `context` parameter of the `KnHandleCreateUserObjectEx()` and `KnHandleCreateBadge()` functions to define the data to be associated with the callable handle and the transfer of the callable handle, respectively (similar to the user resource context and resource transfer context). The server will be able to receive the pointer to this data when dereferencing the callable handle. (Even though the callable handle is an IPC handle, the kernel puts it into the `base_.self` field of the constant part of an IPC request.)

To reject a client request to create one or more IPC channels, call the `DcmCloseHandle()` function to close the DCM handle that was obtained at step 3. (In this case, a `DcmConnectToEndpoint()` or `DcmConnectToComponent()` function call on the client side will end with the `rcNotConnected` error.) If the client request is accepted, the DCM handle obtained at step 3 must also be closed but only after calling the `DcmAcceptEndpointConnection()` or `DcmAcceptComponentConnection()` function.

Dynamically creating IPC channels on the client side

Dynamic creation of IPC channels on the client side includes the following steps:

1. Create a connect to `DCM` by calling the `DcmInit()` function.

   If a client needs to use an individual endpoint (not a set of endpoints of a component instance), you can call the `DcmConnectAny()` or `DcmConnectAnyEx()` function from the auxiliary API for connect.h clients instead of completing the subsequent steps.

2. Create a DCM handle that enables receipt of notifications about publication and unpublication of endpoints with the defined interface or instances of the component with the defined name.

   To complete this step, call the `DcmSubscribeToEndpoints()` or `DcmSubscribeToComponents()` function. The `DcmSubscribeToEndpoints()` function creates a DCM handle that lets you receive notifications about the publication and unpublication of endpoints with the defined interface, and the `DcmSubscribeToComponents()` function creates a DCM handle that lets you receive notifications about the publication and unpublication of component instances with the defined name.

   Use the optional `endpointName`, `componentName`, `serverName`, and `serverHandle` parameters of the `DcmSubscribeToEndpoints()` and `DcmSubscribeToComponents()` functions to include notification filtering based on the qualified name of the endpoint, qualified name of the component instance, server name, and server-identifying handle, respectively. (The server-identifying handle is the descendant of the handle specified in the `serverHandle` parameter of the `DcmPublishEndpoint()` or `DcmPublishComponent()` function that was called on the server side. The client can receive this handle from another process, for example.) Like the handle, the server name uniquely identifies the server. Therefore, you can define a specific server by specifying the handle that identifies this server or the name of this server.

   In the `clientHandle` parameter of the `DcmSubscribeToEndpoints()` and `DcmSubscribeToComponents()` functions, specify the handle that identifies the client. The client-identifying handle can be the handle of this client process that was obtained by sequentially calling the `KosTaskGetSelf()` and `KosTaskGetHandle()` functions from the task.h API, or the handle that was created by calling the `KnHandleCreateUserObject()` function from the handle_api.h API. (When calling the `KnHandleCreateUserObject()` function, the `OCAP_HANDLE_TRANSFER` and `OCAP_HANDLE_GET_SID` flags must be specified in the `rights` parameter.) The client-identifying handle is also needed to complete step 4.

   After calling the `DcmSubscribeToEndpoints()` or `DcmSubscribeToComponents()` function, a notification queue is generated and begins to also receive notifications about the publication of endpoints or component instances that were published before these functions were called and that meet the filter criteria.

   If you no longer need to receive notifications about the publication and unpublication of endpoints or component instances, the DCM handle that was obtained by calling the `DcmSubscribeToEndpoints()` or `DcmSubscribeToComponents()` function must be closed by calling the `DcmCloseHandle()` function.

3. You can extract a notification about endpoint or component instance publishing from the queue of these notifications by calling the `DcmReadPubQueue()` function.

   In the `subscrHandle` parameter of the `DcmReadPubQueue()` function, specify the DCM handle that was obtained at step 2.

   You can use the optional `outServerHandle`, `outServerName`, and `outNodeName` parameters of the `DcmReadPubQueue()` function to get the server-identifying handle, the server name, and the qualified name of the endpoint or component instance, respectively. (The server-identifying handle is the descendant of the handle that was specified in the `serverHandle` parameter of the `DcmPublishEndpoint()` or `DcmPublishComponent()` function that was called on the server side.) This data can be used to decide whether to make a request to create one or more IPC channels with the server. For example, the handle identifying the server may be used to get information about the server from other processes or to get the SID for this handle to compare it to other server-identifying handles available to the client. The server-identifying handle is also needed to complete step 4.

   The value received via the `outPubStatus` parameter of the `DcmReadPubQueue()` function indicates the publication status of the endpoint or component instance. A notification regarding the unpublication of an endpoint or component instance appears in the queue only if the notification about the publication of that endpoint or component instance was previously retrieved. An endpoint or component instance can be unpublished by the server or as a result of server termination.

4. Create a DCM handle that enables fulfillment of a request to create an IPC channel (or channels) to the server.

   To complete this step, call the `DcmCreateConnection()` function. Use the `nodeType` parameter to specify the name of the relevant endpoint interface or the name of the component whose instance provides the necessary set of endpoints. Use the `nodeName` parameter to define the qualified name of the relevant endpoint or the qualified name of the component instance that provides the necessary set of endpoints.

In the `serverHandle` parameter of the `DcmCreateConnection()` function, specify the handle that identifies the server. This handle can be obtained at step 3 via the `outServerHandle` parameter of the `DcmReadPubQueue()` function or by other means, such as from another process. After creating a DCM handle that enables fulfillment of a request to create an IPC channel(s), the server-identifying handle can be closed.

In the `clientHandle` parameter of the `DcmCreateConnection()` function, specify the handle that identifies the client. You need to specify the handle that was defined in the `clientHandle` parameter of the `DcmSubscribeToEndpoints()` or `DcmSubscribeToComponents()` function at step 2. (Servers can receive descendants of this handle through the optional `outClientHandle` parameter of the `DcmListen()` function.) After creating a DCM handle that enables a request to create one or more IPC channels, the handle that identifies the client can be closed but only if this handle was not obtained by calling the `KosTaskGetHandle()` function from the task.h API. (A handle obtained by calling the `KosTaskGetHandle()` function cannot be closed because the closed handle could then be used to identify other resources, which will lead to errors.)

5. Make a request to create an IPC channel or IPC channels with a server.

   To complete this step, call the `DcmConnectToEndpoint()` or `DcmConnectToComponent()` function. The `DcmConnectToEndpoint()` function makes a request to create an IPC channel with the server to use an endpoint, and the `DcmConnectToComponent()` function makes a request to create IPC channels with the server to use the set of endpoints of a component instance.

   In the `connectionId` parameter of the `DcmConnectToEndpoint()` and `DcmConnectToComponent()` functions, specify the DCM handle that was obtained at step 4. In the `sessionHandlesCnt` parameter of the `DcmConnectToComponent()` function, you need to specify the number of endpoints provided by the component instance, which is the number of endpoints listed in the CDL file.

   As a result of the `DcmConnectToEndpoint()` function call, the client uses the `outSessionHandleDesc` parameter to receive a handle transport container that contains a callable handle for using the endpoint. As a result of the `DcmConnectToComponent()` function call, the client uses the `outSessionHandleDescs` parameter to receive an array of handle transport containers that contain callable handles for using the set of endpoints of the component instance. The order of elements in the array matches the order in which the endpoints are listed in the CDL file. The server can provide only a portion of the endpoints of a component instance. If the server decided not to provide an endpoint, the corresponding handle transport container will contain the `INVALID_HANDLE` value instead of the callable handle.

   To receive the callable handle from the handle transport container, use the `nk_get_handle()` macro that is defined in the header file `sysroot-*-kos/include/nk/types.h` from the KasperskyOS SDK.

   The client uses a callable handle to initialize IPC transport. In addition, the client specifies the `INVALID_RIID` value as the Runtime Implementation Identifier (RIID) in the proxy object initialization function.

   After receiving a callable handle, the DCM handle obtained at step 4 must be closed by calling the `DcmCloseHandle()` function.

Interrupting blocking function calls

To interrupt a blocking call of the `DcmConnectToEndpoint()`, `DcmConnectToComponent()`, `DcmReadPubQueue()` or `DcmListen()` function from another thread, call the `DcmInterruptBlockingCall()` function.

Tracking events

You can use the `DcmSubscribeToEvents*()` and `DcmUnsubscribeFromEvents()` functions together with functions from the notice_api.h API to receive notifications about the following events:

- A server publishes or unpublishes an endpoint or component instance (`DCM_PUBLICATION_CHANGED` event mask flag, DCM handle obtained by calling the `DcmSubscribeToEndpoints()` or `DcmSubscribeToComponents()` function).
- A server receives a client request to create an IPC channel to use an endpoint or IPC channels to use a set of endpoints of a component instance (`DCM_CLIENT_CONNECTED` event mask flag, DCM handle obtained by calling the `DcmPublishEndpoint()` or `DcmPublishComponent()` function).
- A blocking call of the `DcmConnectToEndpoint()`, `DcmConnectToComponent()`, `DcmReadPubQueue()`, or `DcmListen()` function is interrupted (`DCM_BLOCKING_CALL_INTERRUPTED` event mask flag, DCM handle obtained by calling the `DcmCreateConnection()`, `DcmSubscribeToEndpoints()`, `DcmSubscribeToComponents()`, `DcmPublishEndpoint()`, or `DcmPublishComponent()` function).
- A server accepts or rejects a client request to create an IPC channel to use an endpoint or IPC channels to use a set of endpoints of a component instance (`DCM_CLIENT_RELEASED_BY_SERVER` event mask flag, DCM handle obtained by calling the `DcmCreateConnection()` function).

The `DcmSubscribeToEvents*()` and `DcmUnsubscribeFromEvents()` functions let you configure the notification receiver by adding or deleting, respectively, tracked objects identified by DCM handles. The `DcmSubscribeToEvents()` function defines the identifier `DCM_EVENT_ID` for "resource–event mask" entries, and the `DcmSubscribeToEventsEx()` function lets you define your own identifiers for "resource–event mask" entries.

Deleting dynamically created IPC channels

A dynamically created IPC channel will be deleted when closing the callable handle and server IPC handle of this IPC channel.

Deleting a DCM connection

If you no longer need to use `DCM`, you need to call the `DcmFini()` function to delete the connection to `DCM` and thereby free up the resources linked to this connection.

Information about API functions

dcm_api.h functions

Function	Information about the function
`DcmInit()`	Purpose Creates a connection to `DCM`. Parameters N/A Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If there is no IPC channel to `DCM`, the function returns `rcNoCapability`. Additional information Thread-safe function. Non-blocking call.
`DcmFini()`	Purpose Deletes a connection to `DCM`. Parameters N/A Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information Thread-safe function. Non-blocking call.
`DcmSubscribeToEvents()`	Purpose Configures the notification receiver to receive notifications about events related to the object identified by the defined DCM handle. Parameters • [in] `notice` – identifier of the notification receiver. The parameter type is defined in the header file `sysroot-*-kos/include/coresrv/handle/notice_api.h` from the KasperskyOS SDK. • [in] `handle` – DCM handle obtained by calling the `DcmSubscribeToEndpoints()`, `DcmSubscribeToComponents()`, `DcmCreateConnection()`, `DcmPublishEndpoint()`, or `DcmPublishComponent()` function. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`DcmSubscribeToEventsEx()`	Purpose Configures the notification receiver to receive notifications about events related to the object identified by the defined DCM handle. Parameters • [in] `notice` – identifier of the notification receiver. The parameter type is defined in the header file `sysroot-*-kos/include/coresrv/handle/notice_api.h` from the KasperskyOS SDK. • [in] `handle` – DCM handle obtained by calling the `DcmSubscribeToEndpoints()`, `DcmSubscribeToComponents()`, `DcmCreateConnection()`, `DcmPublishEndpoint()`, or `DcmPublishComponent()` function. • [in] `eventId` – ID of the "resource–event mask" entry. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`DcmUnsubscribeFromEvents()`	Purpose Configures the notification receiver to not receive notifications about events related to the object identified by the defined DCM handle. Parameters • [in] `notice` – identifier of the notification receiver. The parameter type is defined in the header file `sysroot-*-kos/include/coresrv/handle/notice_api.h` from the KasperskyOS SDK. • [in] `handle` – DCM handle obtained by calling the `DcmSubscribeToEndpoints()`, `DcmSubscribeToComponents()`, `DcmCreateConnection()`, `DcmPublishEndpoint()`, or `DcmPublishComponent()` function. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`DcmCloseHandle()`	Purpose Closes a DCM handle that was obtained by calling the `DcmSubscribeToEndpoints()`, `DcmSubscribeToComponents()`, `DcmCreateConnection()`, `DcmPublishEndpoint()`, `DcmPublishComponent()`, or `DcmListen()` function from the main API, and the `DcmListenGroupPub()` function from the additional API for servers. Parameters • [in] `handle` – DCM handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`DcmInterruptBlockingCall()`	Purpose Interrupts a blocking call of the `DcmConnectToEndpoint()`, `DcmConnectToComponent()`, `DcmReadPubQueue()`, or `DcmListen()` function. Parameters • [in] `handle` – DCM handle specified when calling the `DcmConnectToEndpoint()`, `DcmConnectToComponent()`, `DcmReadPubQueue()` or `DcmListen()` function. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`DcmSubscribeToEndpoints()`	Purpose Creates a DCM handle that enables receipt of a notification about publishing and unpublishing of endpoints with a specific interface. Parameters • [in] `endpointType` – pointer to the name of the interface of endpoints. The maximum name size (in bytes), while taking into account the terminating zero, corresponds to the value of the `UCoreStringSize` constant defined in the `sysroot-*-kos/include/kl/core/Types.idl` file from the KasperskyOS SDK. • [in,optional] `endpointName` – pointer to the qualified name of the endpoint, or `RTL_NULL` to not use notification filtering based on the qualified name of the endpoint. The maximum name size (in bytes), while taking into account the terminating zero, corresponds to the value of the `UCoreStringSize` constant defined in the `sysroot-*-kos/include/kl/core/Types.idl` file from the KasperskyOS SDK. • [in,optional] `serverName` – pointer to the server name, or `RTL_NULL` to not use notification filtering based on the specific server. The maximum name size (in bytes), while taking into account the terminating zero, corresponds to the value of the `UCoreStringSize` constant defined in the `sysroot-*-kos/include/kl/core/Types.idl` file from the KasperskyOS SDK. • [in,optional] `serverHandle` – handle identifying the server, or `INVALID_HANDLE` to not use notification filtering based on a specific server. • [in] `clientHandle` – handle that identifies the client. • [out] `outSubscrHandle` – pointer to the DCM handle that enables receipt of notifications about the publishing and unpublishing of endpoints with a specific interface. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`DcmSubscribeToComponents()`	Purpose Creates a DCM handle that enables receipt of notifications about the publishing and unpublishing of component instances with specific names. Parameters • [in] `componentType` – pointer to the component name. The maximum name size (in bytes), while taking into account the terminating zero, corresponds to the value of the `UCoreStringSize` constant defined in the `sysroot-*-kos/include/kl/core/Types.idl` file from the KasperskyOS SDK. • [in,optional] `componentName` – pointer to the qualified name of the component instance, or `RTL_NULL` to not use notification filtering based on the qualified name of the component instance. The maximum name size (in bytes), while taking into account the terminating zero, corresponds to the value of the `UCoreStringSize` constant defined in the `sysroot-*-kos/include/kl/core/Types.idl` file from the KasperskyOS SDK. • [in,optional] `serverName` – pointer to the server name, or `RTL_NULL` to not use notification filtering based on the specific server. The maximum name size (in bytes), while taking into account the terminating zero, corresponds to the value of the `UCoreStringSize` constant defined in the `sysroot-*-kos/include/kl/core/Types.idl` file from the KasperskyOS SDK. • [in,optional] `serverHandle` – handle identifying the server, or `INVALID_HANDLE` to not use notification filtering based on a specific server. • [in] `clientHandle` – handle that identifies the client. • [out] `outSubscrHandle` – pointer to the DCM handle that enables receipt of notifications about publishing and unpublishing instances of the component with the defined name. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`DcmReadPubQueue()`	Purpose Extracts a notification about the publishing and unpublishing of an endpoint or component instance from the notification queue. Parameters • [in] `subscrHandle` – DCM handle that was created by calling the `DcmSubscribeToEndpoints()` or `DcmSubscribeToComponents()` function. • [in] `msec` – timeout (in milliseconds) before notifications appearing, or `INFINITE_TIMEOUT` to set an unlimited timeout. The constant `INFINITE_TIMEOUT` is defined in the header file `sysroot-*-kos/include/rtl/rtc.h` from the KasperskyOS SDK. • [out,optional] `outServerHandle` – pointer to the handle identifying the server, or `RTL_NULL` if this handle is not required. • [out,optional] `outServerName` – pointer to the buffer for the server name, or `RTL_NULL` if this name is not required. The maximum name size (in bytes), while taking into account the terminating zero, corresponds to the value of the `UCoreStringSize` constant defined in the `sysroot-*-kos/include/kl/core/Types.idl` file from the KasperskyOS SDK. • [in,optional] `serverNameSize` – size (in bytes) for the server name, or `0` if this name is not required. • [out,optional] `outNodeName` – pointer to the buffer for the qualified name of the endpoint or the component instance, or `RTL_NULL` if this name is not required. The maximum name size (in bytes), while taking into account the terminating zero, corresponds to the value of the `UCoreStringSize` constant defined in the `sysroot-*-kos/include/kl/core/Types.idl` file from the KasperskyOS SDK. • [in,optional] `nodeNameSize` – size (bytes) of the buffer for the qualified name of the endpoint or component instance, or `0` if this name is not required. • [out] `outPubStatus` – pointer to a value that indicates the publication status of the endpoint or component instance: `DcmEndpointPublished` – the endpoint has been published, `DcmEndpointUnpublished` – the endpoint has been unpublished, `DcmComponentPublished` – the component instance has been published, `DcmComponentUnpublished` – the component instance has been unpublished. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If a blocking call was interrupted by calling the `DcmInterruptBlockingCall()` function, `rcAbort` is returned. Additional information Non-blocking call if the `msec` parameter is set to `0`.

Function	Information about the function
`DcmCreateConnection()`	Purpose Creates a DCM handle that enables a request to create one or more IPC channels with a server to use a defined endpoint or set of endpoints of a defined component instance. Parameters • [in] `nodeType` – pointer to the endpoint interface name or the component name. The maximum name size (in bytes), while taking into account the terminating zero, corresponds to the value of the `UCoreStringSize` constant defined in the `sysroot-*-kos/include/kl/core/Types.idl` file from the KasperskyOS SDK. • [in] `nodeName` – pointer to the qualified name of the endpoint or component instance. The maximum name size (in bytes), while taking into account the terminating zero, corresponds to the value of the `UCoreStringSize` constant defined in the `sysroot-*-kos/include/kl/core/Types.idl` file from the KasperskyOS SDK. • [in] `serverHandle` – handle that identifies the server. • [in] `clientHandle` – handle that identifies the client. • [out] `outConnectionId` – pointer to the DCM handle that enables fulfillment of a request to create IPC channels to the server to use a specific endpoint or a set of endpoints with a specific component instance. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`DcmConnectToEndpoint()`	Purpose Makes a request to create an IPC channel with a server to use an endpoint. Parameters • [in] `connectionId` – DCM handle obtained by calling the `DcmCreateConnection()` function. • [in] `msec` – timeout (in milliseconds) for fulfilling a request, or `INFINITE_TIMEOUT` to define an unlimited timeout. The constant `INFINITE_TIMEOUT` is defined in the header file `sysroot-*-kos/include/rtl/rtc.h` from the KasperskyOS SDK. • [out] `outSessionHandleDesc` – pointer to the handle transport container that contains the callable handle for using the endpoint. The type of handle transport container is defined in the header file `sysroot-*-kos/include/nk/types.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If the `msec` parameter is set to `0` and the server did not handle the request to create an IPC channel, the function returns `rcNotReady`. If a blocking call was interrupted by calling the `DcmInterruptBlockingCall()` function, `rcAbort` is returned. If the server rejected the request by calling the `DcmCloseHandle()` function, it returns `rcNotConnected`. Additional information Non-blocking call if the `msec` parameter is set to `0`.

Function	Information about the function
`DcmConnectToComponent()`	Purpose Makes a request to create IPC channels with a server to use a set of endpoints of a component instance. Parameters • [in] `connectionId` – DCM handle obtained by calling the `DcmCreateConnection()` function. • [in] `msec` – timeout (in milliseconds) for fulfilling a request, or `INFINITE_TIMEOUT` to define an unlimited timeout. The constant `INFINITE_TIMEOUT` is defined in the header file `sysroot-*-kos/include/rtl/rtc.h` from the KasperskyOS SDK. • [in] `sessionHandlesCnt` – number of endpoints provided by the component instance. • [out] `outSessionHandleDescs` – pointer to an array of handle transport containers that contain callable handles for using the set of endpoints of the component instance. The number of elements in an array cannot exceed the value of the `HandlesLimit` constant defined in the `sysroot-*-kos/include/kl/core/DCM.idl` file from the KasperskyOS SDK. The type of handle transport container is defined in the header file `sysroot-*-kos/include/nk/types.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If the `msec` parameter is set to `0` and the server did not handle the request to create IPC channels, the function returns `rcNotReady`. If a blocking call was interrupted by calling the `DcmInterruptBlockingCall()` function, `rcAbort` is returned. If the server rejected the request by calling the `DcmCloseHandle()` function, it returns `rcNotConnected`. Additional information Non-blocking call if the `msec` parameter is set to `0`.
`DcmPublishEndpoint()`	Purpose Publishes the endpoint to `DCM`. Parameters • [in] `endpointType` – pointer to the name of the endpoint interface. The maximum name size (in bytes), while taking into account the terminating zero, corresponds to the value of the `UCoreStringSize` constant defined in the `sysroot-*-kos/include/kl/core/Types.idl` file from the KasperskyOS SDK. • [in] `endpointName` – pointer to the qualified name of the endpoint. The maximum name size (in bytes), while taking into account the terminating zero, corresponds to the value of the `UCoreStringSize` constant defined in the `sysroot-*-kos/include/kl/core/Types.idl` file from the KasperskyOS SDK. • [in] `serverHandle` – handle that identifies the server. • [out] `outPubHandle` – pointer to the DCM handle that enables receipt of client requests to create an IPC channel to use a specific endpoint. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`DcmPublishComponent()`	Purpose Publishes a component instance to `DCM`. Parameters • [in] `componentType` – pointer to the component name. The maximum name size (in bytes), while taking into account the terminating zero, corresponds to the value of the `UCoreStringSize` constant defined in the `sysroot-*-kos/include/kl/core/Types.idl` file from the KasperskyOS SDK. • [in] `component name` – pointer to the qualified name of the component instance. The maximum name size (in bytes), while taking into account the terminating zero, corresponds to the value of the `UCoreStringSize` constant defined in the `sysroot-*-kos/include/kl/core/Types.idl` file from the KasperskyOS SDK. • [in] `serverHandle` – handle that identifies the server. • [out] `outPubHandle` – pointer to the DCM handle that enables receipt of client requests to create IPC channels to use the set of endpoints of the defined component instance. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`DcmListen()`	Purpose Receives a client request to create an IPC channel (or channels). Parameters • [in] `pubHandle` – DCM handle created by calling the `DcmPublishEndpoint()` or `DcmPublishComponent()` function. • [in] `msec` – timeout (in milliseconds) for the appearance of a client request, or `INFINITE_TIMEOUT` to define an unlimited timeout. The constant `INFINITE_TIMEOUT` is defined in the header file `sysroot-*-kos/include/rtl/rtc.h` from the KasperskyOS SDK. • [out] `outConnectionId` – pointer to the DCM handle that enables you to accept or reject a client request to create an IPC channel (or channels). • [out,optional] `outClientHandle` – pointer to the handle identifying the client, or `RTL_NULL` if this handle is not required. • [out,optional] `outClientName` – pointer to the buffer for the client name, or `RTL_NULL` if this name is not required. The maximum name size (in bytes), while taking into account the terminating zero, corresponds to the value of the `UCoreStringSize` constant defined in the `sysroot-*-kos/include/kl/core/Types.idl` file from the KasperskyOS SDK. • [in,optional] `clientNameSize` – size (in bytes) of the buffer for the client name, or `0` if this name is not required. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If a blocking call was interrupted by calling the `DcmInterruptBlockingCall()` function, `rcAbort` is returned. Additional information Non-blocking call if the `msec` parameter is set to `0`.

Function	Information about the function
`DcmAcceptEndpointConnection()`	Purpose Accepts a client request to create an IPC channel to use an endpoint. Parameters • [in] `connectionId` – DCM handle obtained by calling the `DcmListen()` function. • [in] `sessionHandleDesc` – handle transport container that contains a callable handle for use of the endpoint. The type of handle transport container is defined in the header file `sysroot-*-kos/include/nk/types.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`DcmAcceptComponentConnection()`	Purpose Accepts a client request to create IPC channels to use a set of endpoints of a component instance. Parameters • [in] `connectionId` – DCM handle obtained by calling the `DcmListen()` function. • [in] `sessionHandlesCnt` – number of endpoints provided by the component instance. • [in] `sessionHandleDescs` – pointer to an array of handle transport containers that contain callable handles for using the set of endpoints of the component instance. The number of elements in an array must not exceed the value of the `HandlesLimit` constant defined in the `sysroot-*-kos/include/kl/core/DCM.idl` file from the KasperskyOS SDK. The type of handle transport container is defined in the header file `sysroot-*-kos/include/nk/types.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Using the auxiliary API for clients (connect.h)

The API is defined in the header file `sysroot-*-kos/include/coresrv/dcm/helpers/connect.h` from the KasperskyOS SDK.

The API allows clients to make requests to create IPC channels to servers to use endpoints with the defined interfaces. To execute a request, it is sufficient to call the `DcmConnectAny ()` or `DcmConnectAnyEx ()` function, while when using the basic API dcm_api.h, this requires a sequence of calls: `DcmSubscribeToEndpoints ()`, `DcmReadPubQueue ()`, `DcmCreateConnection ()`, `DcmConnectToEndpoint()`.

Before calling the `DcmConnectAny()` or `DcmConnectAnyEx()` function, you must create a connection to the `DCM` by calling the `DcmInit()` function from the base API dcm_api.h.

As a result of calling the `DcmConnectAny()` or `DcmConnectAnyEx()` function, the client receives a callable handle for the endpoint in the `outSessionHandle` parameter.

A call of the `DcmConnectAny()` function corresponds to a call of the `DcmConnectAnyEx()` function when the `endpointName` and `serverName` parameters are set to `RTL_NULL`, the `serverHandle` and `clientHandle` parameters are set to `INVALID_HANDLE`, and the `findTimeout` and `connectTimeout` parameters are set to `INFINITE_TIMEOUT`.

Information about API functions is provided in the table below.

Function	Information about the function
`DcmConnectAny()`	Purpose Makes a request to create an IPC channel with a server to use an endpoint with the specified interface. Parameters • [in] `endpointType` – pointer to the name of the endpoint interface. The maximum name size (in bytes), while taking into account the terminating zero, corresponds to the value of the `UCoreStringSize` constant defined in the `sysroot-*-kos/include/kl/core/Types.idl` file from the KasperskyOS SDK. • [out] `outSessionHandle` – pointer to a callable handle for using the endpoint. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`DcmConnectAnyEx()`	Purpose Makes a request to create an IPC channel with a server to use an endpoint with the specified interface. Parameters • [in] `endpointType` – pointer to the name of the endpoint interface. The maximum name size (in bytes), while taking into account the terminating zero, corresponds to the value of the `UCoreStringSize` constant defined in the `sysroot-*-kos/include/kl/core/Types.idl` file from the KasperskyOS SDK. • [in,optional] `endpointName` – pointer to the qualified name of the endpoint, or `RTL_NULL` if any endpoint with the defined interface is suitable for use. The maximum name size (in bytes), while taking into account the terminating zero, corresponds to the value of the `UCoreStringSize` constant defined in the `sysroot-*-kos/include/kl/core/Types.idl` file from the KasperskyOS SDK. • [in,optional] `serverName` – pointer to the server name, or `RTL_NULL` if it doesn't matter which specific server provides the endpoint. The maximum name size (in bytes), while taking into account the terminating zero, corresponds to the value of the `UCoreStringSize` constant defined in the `sysroot-*-kos/include/kl/core/Types.idl` file from the KasperskyOS SDK. • [in,optional] `serverHandle` – handle identifying the server, or `INVALID_HANDLE` if it doesn't matter which specific server provides the endpoint. • [in,optional] `clientHandle` – handle that identifies the client, or `INVALID_HANDLE` to use the handle of the calling process to identify the client. • [in] `findTimeout` – timeout (in milliseconds) for publishing an endpoint, or `INFINITE_TIMEOUT` to define an unlimited timeout. The constant `INFINITE_TIMEOUT` is defined in the header file `sysroot-*-kos/include/rtl/rtc.h` from the KasperskyOS SDK. • [in] `connectTimeout` – timeout for a request to create an IPC channel to the server (in milliseconds), or `INFINITE_TIMEOUT` to set an unlimited timeout. The constant `INFINITE_TIMEOUT` is defined in the header file `sysroot-*-kos/include/rtl/rtc.h` from the KasperskyOS SDK. • [out] `outSessionHandle` – pointer to a callable handle for using the endpoint. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If the `connectTimeout` parameter is set to `0` and the server did not handle the request to create an IPC channel, the function returns `rcNotReady`. If the server rejects the request by calling the `DcmCloseHandle()` function from the base API dcm_api.h, the function returns `rcNotConnected`. Additional information Non-blocking call if the `findTimeout` and `connectTimeout` parameters are set to `0`.

Using the additional server API for servers (pub.h)

The API is defined in the header file `sysroot-*-kos/include/coresrv/dcm/groups/pub.h` from the KasperskyOS SDK.

The API allows servers to publish groups of endpoints. From the server perspective, a group of endpoints is a set of endpoints provided by the server and for which the server may receive client requests to create IPC channels using one DCM handle. A server can publish multiple groups of endpoints. A set of endpoints from different groups published by a server must not overlap.

Information about API functions is provided in the table below.

Publishing a group of endpoints

To publish a group of server-provided endpoints to `DCM`, call the `DcmPublishGroupEndpoints()` function. Use the `endpoints` parameter of the `DcmPublishGroupEndpoints()` function to define the qualified names and the contexts of grouped endpoints, and the names of endpoint interfaces. (The array of structures whose pointer is defined via the `endpoints` parameter can be deleted after publication.) An endpoint context is the data that is used by the server to determine which specific endpoint from the published group is required by the client when the `DcmListenGroupPub()` function is called. You can use a numerical identifier or a pointer to specific data as the endpoint context.

A `DcmPublishGroupEndpoints()` function call creates a DCM handle that enables receipt of client requests to create IPC channels to use endpoints in the specific group with specific interfaces.

Receives a client request to create an IPC channel.

To receive a client request to create an IPC channel to use an endpoint in the published group, call the `DcmListenGroupPub()` function. In the `pubHandle` parameter of the `DcmListenGroupPub()` function, specify the DCM handle that was obtained by calling the `DcmPublishGroupEndpoints()` function. The `outContext` output parameter of the `DcmListenGroupPub()` function contains the pointer to the endpoint context. (The server also receives client requests that were made before the endpoint group was published.)

The `DcmListenGroupPub()` function call creates a DCM handle that enables acceptance or rejection of a client request to create an IPC channel to use the specific endpoint in the group that matches the received endpoint context. To accept a client request, call the `DcmAcceptEndpointConnection()` function from the base API dcm_api.h. To reject a client request, call the `DcmCloseHandle()` function from the base API dcm_api.h.

Unpublishing a group of endpoints

To unpublish a group of endpoints, call the `DcmCloseGroupPubHandle()` function to close the DCM handle that was created when the endpoint was published. If a server terminates, all published groups of server endpoints will also be unpublished.

Interrupting blocking function calls

To interrupt a blocking call of the `DcmListenGroupPub()` function from another thread, call the `DcmInterruptGroupPubBlockingCall()` function.

Tracking events

You can use the `DcmSubscribeToGroupPubEvent()` and `DcmUnsubscribeFromGroupPubEvent()` functions together with functions from the notice_api.h API to receive notifications about the following events:

- A server receives a client request to create an IPC channel to use an endpoint from the group (`DCM_CLIENT_CONNECTED` event mask flag, DCM handle obtained by calling the `DcmPublishGroupEndpoints()` function).
- A blocking call of the `DcmListenGroupPub()` function is interrupted (`DCM_BLOCKING_CALL_INTERRUPTED` event mask flag, DCM handle obtained by calling the `DcmPublishGroupEndpoints()` function).

The `DcmSubscribeToGroupPubEvent()` and `DcmUnsubscribeFromGroupPubEvent()` functions let you configure the notification receiver by adding or deleting, respectively, tracked objects identified by DCM handles. The `DcmSubscribeToGroupPubEvent()` function sets the identifier `DCM_EVENT_ID` for entries of the "resource–event mask" type. Flags of the event mask and the "resource–event mask" entry ID `DCM_EVENT_ID` are defined in the header file `sysroot-*-kos/include/coresrv/dcm/dcm_api.h` from the KasperskyOS SDK.

Information about API functions

pub.h functions

Function	Information about the function
`DcmPublishGroupEndpoints()`	Purpose Publishes a group of endpoints to `DCM`. Parameters • [in] `endpoints` – pointer to the array of structures containing information about grouped endpoints, such as the pointers to the qualified names of endpoints, pointers to the names of endpoint interfaces, and the contexts of endpoints. The maximum names size (in bytes), while taking into account the terminating zero, corresponds to the value of the `UCoreStringSize` constant defined in the `sysroot-*-kos/include/kl/core/Types.idl` file from the KasperskyOS SDK. • [in] `endpointsCount` – number of grouped endpoints. • [in] `serverHandle` – handle that identifies the server. • [out] `outPubHandle` – pointer to the DCM handle that enables receipt of client requests to create IPC channels to use endpoints in a specific group with specific interfaces. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`DcmListenGroupPub()`	Purpose Receives a client request to create an IPC channel for use of an endpoint from a published group. Parameters • [in] `pubHandle` – DCM handle obtained by calling the `DcmPublishGroupEndpoints()` function. • [in] `msec` – timeout (in milliseconds) for the appearance of a client request, or `INFINITE_TIMEOUT` to define an unlimited timeout. The constant `INFINITE_TIMEOUT` is defined in the header file `sysroot-*-kos/include/rtl/rtc.h` from the KasperskyOS SDK. • [out] `outContext` – pointer to the context of an endpoint in the published group. • [out] `outConnectionId` – pointer to the DCM handle that enables you to accept or reject a client request to create an IPC channel. • [out,optional] `outClientHandle` – pointer to the handle identifying the client, or `RTL_NULL` if this handle is not required. • [out,optional] `outClientName` – pointer to the buffer for the client name, or `RTL_NULL` if this name is not required. The maximum name size (in bytes), while taking into account the terminating zero, corresponds to the value of the `UCoreStringSize` constant defined in the `sysroot-*-kos/include/kl/core/Types.idl` file from the KasperskyOS SDK. • [in,optional] `clientNameSize` – size (in bytes) of the buffer for the client name, or `0` if this name is not required. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If a blocking call was interrupted by calling the `DcmInterruptGroupPubBlockingCall()` function, `rcAbort` is returned. Additional information Non-blocking call if the `msec` parameter is set to `0`.
`DcmInterruptGroupPubBlockingCall()`	Purpose Interrupts a blocking call of the `DcmListenGroupPub()` function. Parameters • [in] `pubHandle` – DCM handle specified when calling the `DcmListenGroupPub()` function. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`DcmCloseGroupPubHandle()`	Purpose Closes the DCM handle that was received by calling the `DcmPublishGroupEndpoints()` function. Parameters • [in] `pubHandle` – DCM handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`DcmSubscribeToGroupPubEvent()`	Purpose Configures the notification receiver to receive notifications about events related to the object identified by the defined DCM handle. Parameters • [in] `notice` – identifier of the notification receiver. The parameter type is defined in the header file `sysroot-*-kos/include/coresrv/handle/notice_api.h` from the KasperskyOS SDK. • [in] `pubHandle` – DCM handle obtained by calling the `DcmPublishGroupEndpoints()` function. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`DcmUnsubscribeFromGroupPubEvent()`	Purpose Configures the notification receiver to not receive notifications about events related to the object identified by the defined DCM handle. Parameters • [in] `notice` – identifier of the notification receiver. The parameter type is defined in the header file `sysroot-*-kos/include/coresrv/handle/notice_api.h` from the KasperskyOS SDK. • [in] `pubHandle` – DCM handle obtained by calling the `DcmPublishGroupEndpoints()` function. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Using the additional API for clients (subscr.h)

The API is defined in the header file `sysroot-*-kos/include/coresrv/dcm/groups/subscr.h` from the KasperskyOS SDK.

The API allows clients to receive notifications about the publishing and unpublishing of endpoint groups. From the client perspective, a group of endpoints is a set of endpoints on one server for which clients can use a single DCM handle to receive notifications about the publishing and unpublishing of those endpoints. However, there is a restriction that prevents a server from providing multiple endpoints with one interface. Multiple servers may provide identical groups of endpoints required by a client. A server may provide multiple groups of endpoints required by a client. A set of endpoints from different groups required by a client may overlap.

Information about API functions is provided in the table below.

Creating a DCM handle that enables receipt of notifications about the publication and unpublication of groups of endpoints

To create a DCM handle that enables receipt of notifications about the publishing and unpublishing of endpoints with specific interfaces, call the `DcmSetGroupSubscription()` function. Use the `endpointTypes` parameter to define the names of interfaces of grouped endpoints. In the `endpointsCount` parameter, specify a number of grouped endpoints equal to the number of interfaces defined via the `endpointTypes` parameter. (Notifications will contain a number of endpoints equal to the number of interfaces because of the restriction dictating that one separate endpoint corresponds to each interface.)

After calling the `DcmSetGroupSubscription()` function, a notification queue is generated and begins to also receive notifications about the publication of endpoint groups that were published before this function was called and that meet the filter criteria.

Receiving notifications about the publication and unpublication of groups of endpoints

To extract a notification about the publishing or unpublishing of an endpoint group from the notification queue, call the `DcmReadGroupPubQueue()` function. In the `subscrHandle` parameter of the `DcmReadGroupPubQueue()` function, specify the DCM handle that was obtained by calling the `DcmSetGroupSubscription()` function. In the `maxCount` parameter of the `DcmReadGroupPubQueue()` function, specify the number of elements in the array defined via the `outEndpoints` parameter. The value of this parameter must be greater than or equal to the number of endpoints in the group. The value obtained via the `outPubStatus` parameter of the `DcmReadGroupPubQueue()` function indicates that the group of endpoints was published or unpublished. A notification about the publishing of an endpoint group appears in the queue only if the server publishes all endpoints in the group together or one by one. If the server publishes only a portion of the endpoints required by the client, a notification about the publishing of an endpoint group does not appear. A notification regarding the unpublication of an endpoint group appears in the queue only if the notification about the publication of this endpoint group was previously retrieved. An endpoint group can be unpublished by the server or as a result of server termination.

Using the names of interfaces and the qualified names of endpoints resulting from a `DcmReadGroupPubQueue()` function call, you must create DCM handles that allow you to make requests to create IPC channels with the server to use the necessary endpoints. To create these handles, use the `DcmCreateConnection()` function from the base API dcm_api.h. Then you need to make requests to create IPC channels with the server by using the `DcmConnectToEndpoint()` function from the base API dcm_api.h.

Closing a DCM handle that enabled receipt of notifications about the publication and unpublication of groups of endpoints

If you no longer need to receive notifications about the publishing and unpublishing of an endpoint group with specific interfaces, the DCM handle that was created by calling the `DcmSetGroupSubscription()` function must be closed by calling the `DcmCloseGroupSubscrHandle()` function.

Interrupting blocking function calls

To interrupt a blocking call of the `DcmReadGroupPubQueue()` function from another thread, call the `DcmInterruptGroupSubscrBlockingCall()` function.

Tracking events

You can use the `DcmSubscribeToGroupSubscrEvent()` and `DcmUnsubscribeFromGroupSubscrEvent()` functions together with functions from the notice_api.h API to receive notifications about the following events:

- A server publishes or unpublishes an endpoint from a group (`DCM_PUBLICATION_CHANGED` event mask flag, DCM handle obtained by calling the `DcmSetGroupSubscription()` function).
- A blocking call of the `DcmReadGroupPubQueue()` function is interrupted (`DCM_BLOCKING_CALL_INTERRUPTED` event mask flag, DCM handle obtained by calling the `DcmSetGroupSubscription()` function).

The `DcmSubscribeToGroupSubscrEvent()` and `DcmUnsubscribeFromGroupSubscrEvent()` functions let you configure the notification receiver by adding or deleting, respectively, tracked objects identified by DCM handles. The `DcmSubscribeToGroupSubscrEvent()` function sets the identifier `DCM_EVENT_ID` for entries of the "resource–event mask" type. Flags of the event mask and the "resource–event mask" entry ID `DCM_EVENT_ID` are defined in the header file `sysroot-*-kos/include/coresrv/dcm/dcm_api.h` from the KasperskyOS SDK.

Information about API functions

subscr.h functions

Function	Information about the function
`DcmSetGroupSubscription()`	Purpose Creates a DCM handle that enables receipt of notifications about the publishing and unpublishing of endpoint groups with specific interfaces. Parameters • [in] `endpointTypes` – pointer to the array containing the names of interfaces of the endpoints being grouped. The maximum names size (in bytes), while taking into account the terminating zero, corresponds to the value of the `UCoreStringSize` constant defined in the `sysroot-*-kos/include/kl/core/Types.idl` file from the KasperskyOS SDK. • [in] `endpointsCount` – number of grouped endpoints. It must match the number of interfaces defined via the `endpointTypes` parameter. • [in,optional] `serverName` – pointer to the server name, or `RTL_NULL` to not use notification filtering based on the specific server. The maximum name size (in bytes), while taking into account the terminating zero, corresponds to the value of the `UCoreStringSize` constant defined in the `sysroot-*-kos/include/kl/core/Types.idl` file from the KasperskyOS SDK. • [in,optional] `serverHandle` – handle identifying the server, or `INVALID_HANDLE` to not use notification filtering based on a specific server. • [in] `clientHandle` – handle that identifies the client. • [out] `outSubscrHandle` – pointer to the DCM handle that enables receipt of notifications about the publishing and unpublishing of endpoint groups with specific interfaces. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`DcmReadGroupPubQueue()`	Purpose Extracts a notification about the publishing or unpublishing of an endpoint group from the notification queue. Parameters • [in] `subscrHandle` – DCM handle that was obtained by calling the `DcmSetGroupSubscription()` function. • [in] `msec` – timeout (in milliseconds) before notifications appearing, or `INFINITE_TIMEOUT` to set an unlimited timeout. The constant `INFINITE_TIMEOUT` is defined in the header file `sysroot-*-kos/include/rtl/rtc.h` from the KasperskyOS SDK. • [in] `maxCount` – number of elements in the array defined via the `outEndpoints` parameter. It must be greater than or equal to the number of endpoints in the group. • [out,optional] `outServerHandle` – pointer to the handle identifying the server, or `RTL_NULL` if this handle is not required. • [out,optional] `outServerName` – pointer to the buffer for the server name, or `RTL_NULL` if this name is not required. The maximum name size (in bytes), while taking into account the terminating zero, corresponds to the value of the `UCoreStringSize` constant defined in the `sysroot-*-kos/include/kl/core/Types.idl` file from the KasperskyOS SDK. • [in] `serverNameSize` – size (in bytes) for the server name, or `0` if this name is not required. • [out] `outEndpoints` – pointer to the array of structures containing the qualified names of endpoints in the group and the names of interfaces of the group endpoints. The maximum names size (in bytes), while taking into account the terminating zero, corresponds to the value of the `UCoreStringSize` constant defined in the `sysroot-*-kos/include/kl/core/Types.idl` file from the KasperskyOS SDK. • [out] `outEndpointsCount` – pointer to the number of endpoints in the group. • [out] `outPubStatus` – pointer to the value indicating that the group of endpoints was published (`DcmEndpointPublished`) or unpublished (`DcmEndpointUnpublished`). The data type for storing this value is defined in the header file `sysroot-*-kos/include/coresrv/dcm/dcm_api.h`. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If a blocking call was interrupted by calling the `DcmInterruptGroupSubscrBlockingCall()` function, `rcAbort` is returned. Additional information Non-blocking call if the `msec` parameter is set to `0`.
`DcmInterruptGroupSubscrBlockingCall()`	Purpose Interrupts a blocking call of the `DcmReadGroupPubQueue()` function. Parameters • [in] `subscrHandle` – DCM handle specified when calling the `DcmReadGroupPubQueue()` function. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`DcmCloseGroupSubscrHandle()`	Purpose Closes the DCM handle that was obtained by calling the `DcmSetGroupSubscription()` function. Parameters • [in] `subscrHandle` – DCM handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`DcmSubscribeToGroupSubscrEvent()`	Purpose Configures the notification receiver to receive notifications about events related to the object identified by the defined DCM handle. Parameters • [in] `notice` – identifier of the notification receiver. The parameter type is defined in the header file `sysroot-*-kos/include/coresrv/handle/notice_api.h` from the KasperskyOS SDK. • [in] `subscrHandle` – DCM handle that was obtained by calling the `DcmSetGroupSubscription()` function. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`DcmUnsubscribeFromGroupSubscrEvent()`	Purpose Configures the notification receiver to not receive notifications about events related to the object identified by the defined DCM handle. Parameters • [in] `notice` – identifier of the notification receiver. The parameter type is defined in the header file `sysroot-*-kos/include/coresrv/handle/notice_api.h` from the KasperskyOS SDK. • [in] `subscrHandle` – DCM handle that was obtained by calling the `DcmSetGroupSubscription()` function. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

LogRR component

The `LogRR` component provides a system for logging information about program operation. The component includes a server program to which other programs forward messages about their state, and auxiliary output channels used for logging. Messages are sent to the server by using the `logrr_clog` library (the `logrr_cpp` library is available for C++), which filters messages according to the log level and sends them to the server. The server relays messages to output channels. An output channel saves log messages to files.

In the SDK, the LogRR program APIs are provided as static and dynamic client libraries that are linked to the programs whose operating information is logged:

- The `logrr_clog` library and the header file `sysroot-*-kos/include/component/logrr/clog/clog.h` are used in C-language programs.
- The `logrr_cpp` library and the header file `sysroot-*-kos/include/component/logrr/cpp/logger.h` are used in C++ programs.

Adding a log server and output channels to a solution

The `LogrrServer` program sends messages to one or more output channels. You can write log messages to files by using the output channel named `FsOutputChannel`.

If you do not need to centrally process and store logs, you can skip adding a log server and output channels to your solution and instead use the LogRR component API for logging. In this case, you will be able to add a log server and output channels at any time if you will need them.

Adding and configuring logging server

The command for creating and configuring the log server `create_logrr_server()` may contain the following parameters:

`LOG_LEVEL`
: Sets the specified log level. The available values for `LOG_LEVEL` are `CRITICAL`, `ERROR`, `WARNING`, `INFO`, `DEBUG` and `TRACE`. By default, the log server handles messages with the Info level.

`CONFIG_PATH`
: Specifies the path to the configuration file of the log server.

`RECV_CORE_LOGS`
: Enables collection of kernel logs.

`OUTPUT_CHANNELS`
: Connects one or more specified output channels to the server.

`DENY_DYNAMIC_CONN_TYPES`
: Enables rejection of dynamic connections to the log server.

`DENY_STATIC_CONN_TYPES`
: Enables rejection of static connections to the log server.

`VFS_ENTITY`
: Name of the VFS CMake target to which the log server will be connected. Optional parameter.

`VFS_ENTITY_TASK_NAME`
: Name of the system VFS program to which the log server will be dynamically connected. Optional parameter. Use of this parameter excludes the capability to use the `--device` parameter.

`VFS_CONN_ID`
: ID of the static IPC connection to the VFS that the log server will use. Optional parameter. Excludes the capability to use the `VFS_ENTITY_TASK_NAME` parameter.

To add the `LogrrServer` program to a solution:

Edit the root file containing the CMake commands for building the solution or the file containing the `CMake` commands for building the solution image:

CMakeLists.txt

```
include (logrr_server/configure)
include (logrr_fs_output_channel/configure)
# ...
set (TASKS
    # ...
    ClientOne)
# ...
# Add the server that uses the output channel FsOutputChannel
create_logrr_server(
    LOG_LEVEL DEBUG
    OUTPUT_CHANNELS
        logrr.FsOutputChannel)
# Adding a server and output channel to a solution
list(APPEND TASKS
    logrr_server::default_entity
    logrr_fs_output_channel::default_entity)
# Create static IPC channels for source programs to connect to the server
connect_to_logrr_server (ENTITIES ClientOne)
```

Adding and configuring output channels

The command for creating and configuring the channel for printing logs to files `create_logrr_fs_output_channel()` may contain the following parameters:

`CONFIG_PATH`

Specifies the path to the configuration file of the output channel.

`SERVER_TARGET`

Specifies the log server to which the created output channel must connect.

`LOG_DIR`

Specifies the directory for storing log files.

`VFS_ENTITY`

Name of the VFS CMake target to which the log server will be connected. Optional parameter.

`VFS_ENTITY_TASK_NAME`

Name of the system VFS program to which the log server will be dynamically connected. Optional parameter. Use of this parameter excludes the capability to use the `--device` parameter.

`VFS_CONN_ID`

ID of the static IPC connection to the VFS that the log server will use. Optional parameter. Excludes the capability to use the `VFS_ENTITY_TASK_NAME` parameter.

To add the `FsOutputChannel` to a solution:

Edit the root file containing the CMake commands for building the solution or the file containing the CMake commands for building the `Einit` initializing program:

CMakeLists.txt

```
include (logrr_fs_output_channel/configure)
# Add the FsOutputChannel that writes to the file
```

```
create_logrr_fs_output_channel (
    LOG_DIR "/log")
```

Creating IPC channels and allowing interaction between programs

The build system automatically creates IPC channels required for the log system to work, and generates a security policy description that allows interaction between processes.

To create a static connection to the log server, you can use the CMake command `connect_to_logrr_server (ENTITIES <names_of_programs_to_connect>)`.

Getting log entries when running in QEMU

The client libraries `logrr_clog` and `logrr_cpp` send messages to the kernel, which forwards them to the UART port. You can forward data from an emulated UART interface to a standard I/O stream by specifying the `-serial stdio` flag when starting QEMU. In this case, log messages will be displayed in the terminal in which QEMU is running.

Specify the QEMU startup flags in the value of the QEMU_FLAGS parameter that is passed to the `generate_qemu_target ()` command:

CMakeLists.txt

```
generate_qemu_target (qemu
            # ...
            QEMU_FLAGS "-m 4096 -serial stdio"
            # ...
            )
```

Using macros to send messages to a log

The `LogRR` program provides macros for quickly accessing logging functions with a specific log level.

Example use of macros in a C-language program:

C

```
#include <component/logrr/clog/clog.h>
// ...
CLOG(CRITICAL, "Critical message. Exit code = %d", 404);
CLOG(ERROR, "Error message.  Error code = %d", 13);
CLOG(WARNING, "Warning message. Details: %s", "operation not permitted.");
CLOG(INFO, "Info message. Status: %s", "connected.");
CLOG(DEBUG, "Debug message.");
CLOG(TRACE, "Trace message.");
```

To use these macros, you need to link the program to the logrr_clog library.

Example use of macros in a C++ program:

```C++
#include <component/logrr/cpp/logger.h>
// ...
LOG(CRITICAL, "Critical message. Exit code = {}", 404);
LOG(ERROR, "Error message. Error code = {}", 13);
LOG(WARNING, "Warning message. Details: {}", "operation not permitted.");
LOG(INFO, "Info message. Status: {}", "connected.");
LOG(DEBUG, "Debug message.");
LOG(TRACE, "Trace message.");
```

To use these macros, you need to link the program to the logrr_cpp library.

Log levels

A log level is an attribute of each message in the logging system. The log level is configured through the server and is used when filtering messages in a client.

Critical

Critical malfunction in program operation that makes it impossible for the program to continue working.

Error

Program operating error that does not completely prevent the program from continuing to work. This level is applied for logging an unexpected termination of the current operation. It is not recommended to apply this level for logging erroneous user actions.

Warning

Problem situation that is anticipated and normally handled by the program.

Info (default)

Information about the current functional performance of the program without additional details. For example, this info can include switching to a new window in the user interface or successfully writing an entry in the database.

Debug

More detailed log level than Info. It provides more details on program events and may be useful for debugging.

Trace

The most detailed log level that is used for the most detailed tracking of program events. When this level is enabled, it can have a substantial impact on performance.

Each successive level includes all previous levels. For example, if a server is configured for the Warning level, it forwards messages with the Critical, Error and Warning levels to the log while ignoring messages with the Info, Debug and Trace levels.

The log level when the program sends a message can be defined by doing the following:

- Using the macro with the parameter corresponding to the selected level.
- Passing the value of the `LogrrLogLevel` or `LogLevel` enumerator when logging functions are called.

LogRR program API

The libraries of the `LogRR` program provide APIs for sending messages to the log through the log server and APIs for managing the log server.

logrr_clog library

The `logrr_clog` library provides the API for sending messages to the log.

> The `logrr_clog` library is intended for use in C-language programs. Use the `logrr_cpp` library in C++ programs.

To get access to the library API:

1. In the CMake build instructions, include the `logrr_clog` package by running the `find_package()` command.

   ```
   find_package(logrr_clog REQUIRED)
   ```

2. Use the `target_link_libraries()` command to link your build target to the library. To get the name of the library, use the `logrr_clog::logrr_clog` target that is provided by the `logrr_clog` package.

   ```
   target_link_libraries (${TARGET_NAME} logrr_clog::logrr_clog)
   ```

   The `logrr_clog::logrr_clog` target links to the dynamic library by default. Also, you can link to the dynamic library explicitly by using the `logrr_clog::logrr_clog-shared` target. If you want to link to a static library, use the `logrr_clog::logrr_clog-static` target.

ClogInit(), ClogLog() functions and CLOG macro

The `ClogInit()` function Initializes the client side of the `LogRR` component. The method accepts the ID of the log provider. If the `ClogInit()` function is not explicitly called, initialization will be performed when the `CLOG` macro is called for the first time, and the name of the program that called the macro will be set as the provider ID.

The `ClogLog()` function writes to the log.

> The `ClogLog()` function is intended for use in C-language programs. In C++ programs, use the Logger::Log() function or LOG_* macros.

A description of the `ClogInit()`, `ClogLog()` functions and macros for quick access to it are provided in the file `/opt/KasperskyOS-Community-Edition-<platform>-<version>/sysroot-*-kos/include/component/logrr/clog/clog.h`.

Parameters of the `ClogLog()` function:

`level`
: Log level `LogrrLogLevel`.

`file`
: File name.

`line`
: Number of the line in the file.

`func`
: Function name.

`message`
: Message text or message text formatting string.

`...`
: Parameters to be inserted into the `message` formatting string.

Macro for quick access to the ClogLog() function

Instead of calling the `ClogLog()` function, you can use the macro whose description is provided in the file `component/logrr/clog/clog.h`. This macros is variadic (takes a variable number of parameters), which lets you avoid specifying all parameters of the `ClogLog()` function. When calling this macro, you only need to specify the log level and message text or the message formatting string with the values of parameters. The applied log `level` is determined by the first parameter of the macro. An example of using this macro is provided in the section titled Using macros to send messages to a log.

LogrrLogLevel enumerated type

The enumerated type `LogrrLogLevel` contains fields whose names correspond to the log levels available in the `LogRR` program.

> The enumerated type `LogrrLogLevel` is intended for use in C-language programs. In C++ programs, use the `LogLevel` class.

`LogrrLogLevel` enumerators can be passed to the `ClogLog()` function to send a message with the specified log level.

A description of `LogrrLogLevel` enumeration is provided in the file `/opt/KasperskyOS-Community-Edition-<platform>-<version>/sysroot-*-kos/include/component/logrr/core/log_level_c.h`.

component/logrr/core/log_level_c.h (snippet)

```
typedef enum LogrrLogLevel
{
    LogrrLogLevel_Critical, /*!< Critical errors when process can't continue to work.  */
    LogrrLogLevel_Error,    /*!< Some issues that allow to continue application execution. */
    LogrrLogLevel_Warning,  /*!< Some regular errors that application can handle. */
```

```
    LogrrLogLevel_Info,     /*!< Some useful information about application work/state. */
    LogrrLogLevel_Debug,    /*!< Detailed log to understand/fix some specific issues. */
    LogrrLogLevel_Trace,    /*!< To "trace" some function execution. May affect performance. */
} LogLevel;
```

logrr_cpp library

The `logrr_cpp` library provides the API for sending messages to the log.

> The `logrr_cpp` library is intended for use in C++ programs. In C-language programs, use the `logrr_clog` library.

To get access to the library API:

1. In the CMake build instructions, include the `logrr_cpp` package by running the `find_package()` command.

   ```
   find_package(logrr_cpp REQUIRED)
   ```

2. Use the `target_link_libraries()` command to link your build target to the library. To get the name of the library, use the `logrr_cpp::logrr_cpp` target that is provided by the `logrr_cpp` package.

   ```
   target_link_libraries (${TARGET_NAME} logrr_cpp::logrr_cpp)
   ```

   The `logrr_cpp:logrr_cpp` target links to the dynamic library by default. Also, you can link to the dynamic library explicitly by using the `logrr_cpp::logrr_cpp-shared` target. If you want to link to a static library, use the `logrr_cpp::logrr_cpp-static` target.

Init(), Log() methods and LOG macro

The `logrr::Init()` method initializes the client part of the `LogRR` component. The method accepts the ID of the log provider. If the `logrr::Init()` method is not explicitly called, initialization will be performed when the `LOG` macro is called for the first time, and the name of the program that called the macro will be set as the provider ID.

The `logrr::Log()` method writes to the log.

Method parameters:

`logLevel`
: Log level `LogLevel`.

`sourceLocation`
: Object that determines the location in the source code (file name, function name, line number and column number). The `std::source_location` class is used in C++20, while the `std::experimental::source_location` class is used in earlier versions.

`format`
: Message text formatting string.

`args`

Parameters to be inserted into the `format` formatting string.

> The `logrr::Log()` method is intended for use in C++ programs. In C-language programs, use the static function ClogLog() or CLOG macro.

A description of the `logrr::Init()` and `logrr::Log()` methods and macros for quick access to logging method are provided in the file `/opt/KasperskyOS-Community-Edition-<platform>-<version>/sysroot-*-kos/include/component/logrr/cpp/logger.h`.

component/logrr/cpp/logger.h (snippet)

```
namespace logrr {

__rtl_public
void Init(const char* providerName = nullptr) noexcept;

__rtl_public
bool ShouldLog(logrr::LogLevel logLevel, logrr::SubProviderId id, std::string_view filename = std::string_view{})
noexcept;

__rtl_public
void RegisterSubProvider(std::string_view name, logrr::SubProviderId* id) noexcept;

__rtl_public
void LogWithClog(
    logrr::SubProviderId subProviderId,
    LogLevel logLevel,
    std::string_view file,
    const uint32_t line,
    std::string_view function,
    std::string_view msg) noexcept;

namespace details {

__rtl_public
void PrintErrorMessage(std::string_view fileName, std::uint32_t line, std::string_view exMsg) noexcept;

__rtl_public
void PrintWarnMessage(std::string_view fileName, std::uint32_t line, std::string_view exMsg) noexcept;

template <bool IsSecure, typename... Args>
__rtl_public
void LogImpl(
    logrr::SubProviderId subProviderId,
    logrr::LogLevel logLevel,
    std::string_view file,
    const uint32_t line,
    std::string_view function,
    fmt::format_string<Args...> format,
    Args&&... args)
{
    constexpr std::size_t MaxLength{LOGRR_LOG_MESSAGE_MAX_SIZE - 1};

    if constexpr (sizeof...(Args) == 0)
    {
        std::string_view msg(format.get().data(), format.get().size());
        LogWithClog(subProviderId, logLevel, file, line, function, msg);
    }
    else
    {
        constexpr std::size_t BufInitialSize{2 * LOGRR_LOG_FRAGMENT_MAX_SIZE};

        std::string buf;
        buf.reserve(BufInitialSize);

        auto result = fmt::vformat_to_n(
            std::back_inserter(buf),
            MaxLength,
            format,
            fmt::make_format_args(args...));
```

```
        if (result.size > MaxLength)
        {
            details::PrintWarnMessage(file, line,
                fmt::format("Log message was truncated to {} bytes", MaxLength));
        }

        buf.push_back('\0');

        if constexpr (IsSecure)
        {
            SecretMaskingContext::GetMasker()->ApplyMasking(buf);
        }

        std::string_view formattedStr{buf.data(), buf.size()};

        LogWithClog(subProviderId, logLevel, file, line, function, formattedStr);
    }
}

} // namespace details

template <typename SourceLocationType, typename... Args>
__rtl_public
void Log(
    logrr::SubProviderId subProviderId,
    logrr::LogLevel logLevel,
    SourceLocationType sourceLocation,
    fmt::format_string<Args...> format,
    Args&&... args) noexcept
{
    Log(subProviderId, logLevel, sourceLocation.file_name(), sourceLocation.line(),
        sourceLocation.function_name(), format, std::forward<Args>(args)...);
}

template <typename... Args>
__rtl_public
void Log(
    logrr::SubProviderId subProviderId,
    logrr::LogLevel logLevel,
    std::string_view file,
    const uint32_t line,
    std::string_view function,
    fmt::format_string<Args...> format,
    Args&&... args) noexcept
try
{
    details::LogImpl<false>(
        subProviderId, logLevel, file, line, function,
        format, std::forward<Args>(args)...);
}
catch (const std::exception& e)
{
    details::PrintErrorMessage(file, line, e.what());
}

} // namespace logrr

#define __CPPLOG_LEVEL_TRACE__    logrr::LogLevel::Trace
#define __CPPLOG_LEVEL_DEBUG__    logrr::LogLevel::Debug
#define __CPPLOG_LEVEL_INFO__     logrr::LogLevel::Info
#define __CPPLOG_LEVEL_WARNING__  logrr::LogLevel::Warning
#define __CPPLOG_LEVEL_ERROR__    logrr::LogLevel::Error
#define __CPPLOG_LEVEL_CRITICAL__ logrr::LogLevel::Critical

#define CALL_LOG_SL(level, file, line, func, ...) \
    do { \
        if (::logrr::ShouldLog(level, *LogrrSubProviderId, file)) \
            ::logrr::Log(*LogrrSubProviderId, level, file, line, func, __VA_ARGS__); \
    } while (0)

#define CALL_LOG(level, ...) \
    do { \
        const auto sl = logrr::SourceLocation::current(); \
        CALL_LOG_SL(level, sl.file_name(), sl.line(), sl.function_name(), __VA_ARGS__); \
    } while (0)

#define LOG(level, ...) CALL_LOG(__CPPLOG_LEVEL_ ## level ## __, __VA_ARGS__)
#define LOG_SL(level, file, line, func, ...) CALL_LOG_SL(__CPPLOG_LEVEL_ ## level ## __, file, line, func,
__VA_ARGS__)
```

A macro for quick access to the Log() method

Instead of calling the static method `Logger::Log()`, you can use the macro whose description is provided in the file `component/logrr/cpp/logger.h`. This macro is variadic (takes a variable number of parameters), which lets you avoid specifying all parameters of the `Logger::Log()` method. When calling this macro, you only need to specify the log level and message formatting string with the values of parameters. The applied log level `logLevel` is determined by the first parameter of the macro: An example of using these macros is provided in the section titled Using macros to send messages to a log.

LogLevel class

The `LogLevel` class is an enumeration (`enum class`) and contains fields whose names correspond to the log levels available in the `LogRR` program.

> The `LogLevel` class is intended for use in C++ programs. In C-language programs, use the enumerated type `LogrrLogLevel`.

Values of the `LogLevel` enumeration can be passed to the static `Log()` method of the `Logger` class.

A description of the `LogLevel` class is provided in the file `/opt/KasperskyOS-Community-Edition-<platform>-<version>/sysroot-*-kos/include/component/logrr/core/log_level.h`.

component/logrr/core/log_level.h (snippet)

```
enum class LogLevel : int8_t
{
    Critical = 0,
    Error,
    Warning,
    Info,
    Debug,
    Trace
};
```

ExecutionManager component

The ExecutionManager component provides interfaces for managing the lifecycle of processes through IPC mechanisms and lets you do the following:

- Start a process.
- Get information about a process.
- Stop a process.
- Shut down the system.

The ExecutionManager component usage scenario and description are provided in Starting processes using the system program ExecutionManager.

The ExecutionManager component's API is defined in the header files located in the directory `sysroot-*-kos/include/component/execution_manager/` from the KasperskyOS Community Edition SDK.

execution_manager_proxy.h interface

The API is defined in the header file `sysroot-*-kos/include/component/execution_manager/kos_ipc/execution_manager_proxy.h` from the KasperskyOS Community Edition SDK.

The interface contains functions for getting the pointer to the instance of the `IExecutionManager` interface that is required for working with the ExecutionManager component.

Information about API functions is provided in the table below.

The API provides the following two ways to obtain a pointer to the `IExecutionManager` interface by connecting to the ExecutionManager process using static methods of the `ExecutionManagerConnector` object:

- Statically create an IPC channel in the `StaticConnect()` function.
- Dynamically create an IPC channel in the `DcmConnect()` function. This method uses the `DCM` system program.

Using the `cfg` input parameter, the `CreateExecutionManager()` function accepts configuration parameters in the `ExecutionManagerConfig` structure. All fields of this structure are optional.

```
struct ExecutionManagerConfig
{
        // Name of the IPC channel for connecting to the ExecutionManager process
    char const* const mainConnection            = KOS_EXECMGR_CONN_MAIN;
    // Name of the endpoint that implements the IApplicationController interface
    char const* const appControlInterface       = KOS_EXECMGR_IFACE_AC;
    // Name of the endpoint that implements the IStateProvider interface
    char const* const appStateInterface         = KOS_EXECMGR_IFACE_AS;
    // Name of the endpoint that implements the ISystemController interface
    char const* const systemControlInterface    = KOS_EXECMGR_IFACE_SC;
    // Name of the endpoint that implements the IProcessControl interface
    char const* const processControlInterface   = KOS_EXECMGR_IFACE_PC;
    // ExecutionManager process class name
    char const* const execmgrServerName         = KOS_EXECMGR_SERVER_NAME;
};
```

execution_manager_proxy.h functions

Function	Information about the function
`StaticConnect()`	Purpose Gets the pointer to an instance of the `IExecutionManager` interface using a static IPC channel to connect to the ExecutionManager process. Parameters • [in] `connName` – name of the IPC channel. Returned values: Pointer to the instance of the `IExecutionManager` interface.

Function	Information about the function
`DcmConnect()`	Purpose Gets the pointer to an instance of the `IExecutionManager` interface using the system program `DCM` to connect to the ExecutionManager process. Parameters • [in] `timeout` – timeout (in milliseconds) for creating a connection to the name server. Default value: `30` sec. Returned values: Pointer to the instance of the `IExecutionManager` interface.
`CreateExecutionManager()`	Purpose Gets the pointer to the instance of the `IExecutionManager` interface that is required for working with the ExecutionManager component. Obsolete function; you are advised to use other functions from this table. Parameters • [in] `cfg` – `ExecutionManagerConfig` structure containing the configuration settings for the connection to the `ExecutionManager` process. • [out] `emp` – pointer to the instance of the `IExecutionManager` interface. Returned values: If successful, the function returns `kos::rtl::Ok`, otherwise it returns an error code.

Usage example:

client.cpp

```
#include <component/execution_manager/kos_ipc/execution_manager_proxy.h>

int main(int argc, const char *argv[])
{
    // ...
    execmgr::IExecutionManagerPtr ptr = execution_manager::ipc::ExecutionManagerConnector::DcmConnect();
    // ...
}
```

IExecutionManager interface

The API is defined in the header file `sysroot-*-kos/include/component/execution_manager/i_execution_manager.h` from the KasperskyOS Community Edition SDK.

The `IExecutionManager` interface lets you access pointers to the following interfaces:

- `IProcessControl` – interface that lets you start a process from an executable file installed by the PackageManager component from the KPA package in a KasperskyOS-based solution, and lets you receive information about the process (including information about its completion) and stop this process.
- `IApplicationController` – interface that lets you start a process from any executable file whose location in the file system is known, and lets you stop this process.

- `IStateProvider` – interface for receiving information about a process that was started using the `IApplicationController` interface.

- `ISystemController` – interface for receiving information about processes that were started by using the ExecutionManager component, and for shutting down the system.

Usage example:

client.cpp

```
int main(int argc, const char *argv[])
{
    // ...
    alm::execution_manager::IProcessControlPtr pc;
    res = emPtr->GetProcessController(pc);
    if (res != kos::Ok)
    {
        std::cerr << fmt::format("[{}] Failed to create process controller. res=0x{:x}\n", selfName, res);
        return EXIT_FAILURE;
    }

    execmgr::IApplicationControllerPtr ac;
    if (ptr->GetApplicationController(ac) != kos::Ok)
    {
        std::cerr << "Cannot get application controller" << std::endl;
        return EXIT_FAILURE;
    }

    execmgr::ISystemControllerPtr sc;
    if (ptr->GetSystemController(sc) != kos::Ok)
    {
        std::cerr << "Cannot get system controller" << std::endl;
        return EXIT_FAILURE;
    }

    // ...
}
```

IProcessControl interface

The API is defined in the header file `sysroot-*-kos/include/alm/execution_manager/i_process_control.h` from the KasperskyOS SDK.

The API lets you do the following:

- Start a process from an executable file installed by the PackageManager component from the KPA package in a KasperskyOS-based solution.

- Get information about the startup configuration that was used to start the process.

- Get information about process, including information about its termination.

- Stop a process.

Information about API functions is provided in the table below.

Starting a process

To start the process, call the `StartProcess()` function. Using the `opts` input parameter, this function accepts process startup parameters in the `StartOptions` structure. All fields of this structure are optional for initialization.

```
struct StartOptions
{
    // Handle that identifies the parent process.
    IpcHandle       parent{NK_INVALID_HANDLE, 0, NK_INVALID_HANDLE};
    // List of custom handles
    // that will be transferred to the process when it is started.
    UserHandleItems userHandles;
    // Command-line arguments.
    Strings         extraArgs;
    // Environment variables.
    Strings         extraEnvs;
};
```

During initialization of the `StartOptions` structure, the `parent` handle transport container is populated with the default values, which indicate the absence of a parent process. You can create a process hierarchy by passing the handle of a parent process in this parameter to a started child process.

The `userHandles` vector is a component of the `StartOptions` structure and contains a list of custom handles that will be transferred to the process when it is started. Items of this list are pairs that are defined in the `UserHandleItem` structure. These pairs consist of a handle and its name. After a process is started, the handles transferred to this process can be found by using the `KosTaskLookupResource()` function based on the names of these handles.

```
struct UserHandleItem
{
    // Handle name.
    std::string id;
    // Handle.
    IpcHandle   handle;
};
```

The input parameters `pkgId` and `runConfigId` of the `StartProcess()` function are defined in the KPA package manifest. Their values may be received by using functions from the IPackageManifest interface of the PackageManager component.

Stopping a process

To stop the process, call the `StopProcess()` function. Using the `stopPolicy` input parameter, this function receives the process stop policy from the `StopPolicy` enumeration.

```
enum class StopPolicy : uint32_t
{
    // Send a stop signal to the process.
    Normal = 0,
    // Immediately stop the process.
    Forced,
};
```

Receiving information about a process

To get information about a process, call the `GetProcessState()` function. Using the `state` output parameter, this function returns information about a process in the `ProcessContext` structure.

```
struct ProcessContext
{
    // State of the process.
    ProcessState         state{};
    // System status of process termination.
    ExecutableExitStatus exitStatus{};
    // Common code for process termination.
    ExecutableExitCode   exitCode{};
    // Specific code for process termination.
    ExecutableExitReason exitReason{};
};
```

The `state` enumeration is a component of the `ProcessContext` structure and describes the state of a process.

```
enum class ProcessState : uint32_t
{
    // Service constant.
    None = 0,
    // Process created.
    Created,
    // Process initialized.
    Prepared,
    // Process started.
    Running,
    // Process stopped unexpectedly.
    Failed,
    // Process stopped, but data on the process is available
    // to the GetProcessState() function.
    Finished,
    // Process terminated and does not exist in the system.
    NotExisting,
};
```

The `exitStatus` enumeration is a component of the `ProcessContext` structure and describes the reason for stopping a process. The `ExecutableExitStatus` enumeration is defined in the header file `sysroot-*-kos/include/alm/execution_manager/execution_manager_types.h` from the KasperskyOS SDK.

```
enum class [[nodiscard]] ExecutableExitStatus : std::underlying_type_t<TaskExitStatus>
{
// Process has not yet been started by the ExecutionManager component.
    ExitUninitialized = TaskExitStatus::TaskExitUninitialized,
    // Process stopped unexpectedly as a result of an unhandled exception.
    ExitUnexpected    = TaskExitStatus::TaskExitUnexpected,
    // Process terminated on its own, including
    // after receiving a stop signal via a call
    // of the StopProcess() function with the Normal value in the process stop policy.
    ExitNormal        = TaskExitStatus::TaskExitNormal,
    // Process stopped by external request, including via a call of the
    // StopProcess() function with the Forced value in the process stop policy.
    ExitTerminated    = TaskExitStatus::TaskExitTerminated,
};
```

The numeric `exitCode` parameter is a component of the `ProcessContext` structure and contains the return code for a process that stopped on its own, including after receiving a stop signal. The values for this return code are defined by the developer of the KasperskyOS-based solution.

The numeric `exitReason` parameter is a component of the `ProcessContext` structure and contains the return code that specifies the reason for an unsuccessful attempt to start the process, or `rcOk` if the startup was successful. Parameter is defined in the header file `sysroot-*-kos/include/rtl_cpp/retcode.h` from the KasperskyOS SDK. This file contains return codes that are common for the APIs of all solution components and their constituent parts (see Return codes).

Getting information about the startup configuration

To request information about the startup configuration that was used when the process was started, call the `GetRunConfigurationInfo()` function. Using the `info` output parameter, the function returns information about the startup configuration as a `RunConfigurationInfo` structure.

```
struct RunConfigurationInfo
{
    // ID of the KPA package.
    std::string packageId;
    // Startup configuration ID.
    std::string runConfigurationId;
};
```

If you do not need the full startup configuration details and only need to check whether the process was started from the KPA package, call the `IsKpaProcess()` function. This function uses the `isKpa` output parameter to return the result of the check: `true` if the process was started from the KPA package, otherwise `false`.

Failure to execute the `GetRunConfigurationInfo()` function is not a reliable indication that the process was not started from the KPA package. To check this, use the `IsKpaProcess()` function.

i_process_control.h functions

Function	Information about the function
`StartProcess()`	Purpose Resumes a process. Parameters • [in] `pkgId` – ID of the KPA package. • [in] `runConfigId` – ID of the startup configuration. • [in] `opts` – `StartOptions` structure containing the process startup parameters. • [out] `accessToken` – link to the handle that identifies the started process. Returned values: If successful, the function returns `kos::rtl::Ok`, otherwise it returns an error code.
`StopProcess()`	Purpose Stops a process. Parameters • [in] `accessToken` – handle that identifies the started process. • [in] `stopPolicy` – process stop policy. Returned values: If successful, the function returns `kos::rtl::Ok`, otherwise it returns an error code.

Function	Information about the function
`GetProcessState()`	Purpose Gets information about a process. Parameters • [in] `accessToken` – handle that identifies the started process. • [out] `state` – link to the `ProcessContext` structure containing information about the process. Returned values: If successful, the function returns `kos::rtl::Ok`, otherwise it returns an error code.
`GetRunConfigurationInfo()`	Purpose Gets information about the startup configuration that was used to start the process. Parameters • [in] `accessToken` – handle that identifies the started process. • [out] `info` – link to the `RunConfigurationInfo` structure containing information about the startup configuration. Returned values: If successful, the function returns `kos::rtl::Ok`, otherwise it returns an error code.
`IsKpaProcess()`	Purpose Checks whether the process was started from the KPA package. Parameters • [in] `accessToken` – handle that identifies the started process. • [out] `isKpa` – result of the check: `true` if the process was started from the KPA package, otherwise `false`. Returned values: If successful, the function returns `kos::rtl::Ok`, otherwise it returns an error code.

Usage example:

client.cpp

```
int main(int argc, const char *argv[])
{
    // ...

    alm::execution_manager::IProcessControl::StartOptions startOptions{};
    alm::execution_manager::IpcHandle procToken{};

    // ID of the KPA package.
    const std::string   packageId{"application.Application"};
    // Startup configuration ID.
    const std::string   runConfigId{"app"};

    kos::rtl::Result res = pc->StartProcess(packageId, runConfigId, startOptions, procToken);
    if (res != kos::rtl::Ok)
    {
        std::cerr << fmt::format("[{}] Failed to start process. res=0x{:x}\n", selfName, res);
        return EXIT_FAILURE;
    }

    alm::execution_manager::ProcessContext procContext{};

    res = pc->GetProcessState(procToken, procContext);
    if (res != kos::rtl::Ok)
    {
        std::cerr << fmt::format("[{}] Failed to get process state. res=0x{:x}\n", selfName, res);
```

```
        return EXIT_FAILURE;
    }

    if (procContext.state == alm::execution_manager::ProcessState::Running)
    {
        std::cerr << fmt::format("[{}] Process state is Running.\n", selfName);
    }

    res = pc->StopProcess(procToken, alm::execution_manager::StopPolicy::Forced);
    if (res != kos::rtl::Ok)
    {
        std::cerr << fmt::format("[{}] Failed to stop process. res=0x{:x}\n", selfName, res);
        return EXIT_FAILURE;
    }

    // ...
}
```

IApplicationController interface

This API is defined in the header file `sysroot-*-kos/include/component/execution_manager/i_application_control.h` from the KasperskyOS SDK.

The API lets you start a process from an executable file and stop this process.

Information about API functions is provided in the table below.

To start the process, call the `StartEntity()` function. Using the `info` input parameter, this function accepts process startup parameters in the `StartEntityInfo` structure. All fields of this structure are optional for initialization. Using the `resInfo` output parameter, this function returns the link to the `StartEntityResultInfo` structure containing the process startup results.

```
struct IApplicationController
{
    struct StartEntityInfo
    {
        // Process name.Unless otherwise specified, the process class name will be used.
        // If the process class name is not specified, the executable file name will be used.
        std::string              entityName{};
        // Process class. Unless otherwise specified, the process name will be used.
        // If the process name is not specified, the executable file name will be used.
        std::string              eiid{};
        // Command-line arguments.
        std::vector<std::string> args{};
// Environment variables that are passed to the started process.
        std::vector<std::string> envs{};
        // Environment variables that are passed to the isolated data storage of the process.
        std::vector<std::string> containerEnvs{};
        // Flag that defines the behavior of the Execution Manager when the process starts:
        // true – pause execution and wait for the ready event from the
        //         started process, which confirms its readiness to work.
        // false – continue working immediately after starting the process
        //         without waiting for confirmation of its readiness.
        bool                     wait{false};
        // Bitmask of flags that control the behavior of the isolated data storage of the process:
        // 0                              – create a copy of the storage for the started process.
        // EM_FLAG_DO_NOT_COPY_CONTAINER – use the storage of the parent process
        //                                  instead of creating a new copy.
        uint32_t                 flags{0};
        // Thread stack limit in bytes or 0 to use the default size
        // defined when creating the process.
        uint32_t stackSize{0};
        // Policy for handling abnormal termination of the process:
        // CrashMode ::None – do not perform any action.
        // CrashMode ::Dump – create a crash dump.
        // CrashMode ::Panic – shut down or restart the hardware platform.
        CrashMode                mode{CrashMode::None};
    };

    struct StartEntityResultInfo
```

```
    {
        // Security class assigned to the process.
        std::string eiid;
        // Handle that identifies the started process.
        EntityId    entId{nk_handle_desc(INVALID_HANDLE)};
        // Security ID of the started process.
        Uid         sid{0};
        // Name of the started process.
        std::string taskName;
    };
};
```

If the `mode` field of the `StartEntityInfo` structure is set to `Dump`, a crash dump is automatically generated using the `DumpCollector` system program whenever the process crashes. The `DumpCollector` system program must be included in the developed KasperskyOS-based solution. (For more details on the crash dump mechanism, see Dump_collector example.) A crash dump contains diagnostic data, which includes a stack trace, the state of processor registers, thread information, and a process memory map. The generated crash dump is printed to the console using the LogRR component.

To start a process and pass a list of custom handles to it, call the `StartEntityWithUserHandlesEx()` function. This function receives the `userHandles` vector as an input parameter. This vector contains a list of custom handles that are passed to the process when it is started. Vector elements are `UserHandleItem` structures, which consist of a handle and its name. After a process is started, the handles transferred to this process can be found by using the `KosTaskLookupResource()` function based on the names of these handles.

```
struct UserHandleItem
{
    // Handle name.
    std::string id;
    // Handle.
    IpcHandle   handle;
```

};

The `StartEntityWithUserHandles()` function starts a process and passes a list of handles to it. The list of handles is defined in the `userHandles` parameter, which is an array of the `UserHandlesArray` type:

```
using UserHandlesArray = std::array<nk_handle_desc_t, EM_USER_HANDLES_MAX>;
```

To stop a process, call the `ShutdownEntity()` or `StopEntity()` function. The `entId` input parameter is used by these functions to receive the handle that identifies the started process.

i_application_control.h functions

Function	Information about the function
`StartEntity()`	Purpose Resumes a process. Parameters • [in] `runPath` – path to the executable file to be started. • [in] `info` – structure containing the `StartEntityInfo` process startup parameters. • [out] `resInfo` – structure containing the `StartEntityResultInfo` process startup results. • [in] `containerPath` – path to the isolated data storage of a process. Returned values: If successful, the function returns `kos::Ok`, otherwise it returns an error code.

Function	Information about the function
`StartEntityWithUserHandles()`	Purpose Starts a process a passes the list of handles to it. The names of the handles are generated automatically. Parameters • [in] `runPath` – path to the executable file to be started. • [in] `info` – structure containing the `StartEntityInfo` process startup parameters. • [in] `userHandles` – list of handles that are passed to the process when it is started. • [out] `resInfo` – structure containing the `StartEntityResultInfo` process startup results. • [in] `containerPath` – path to the isolated data storage of a process. Returned values: If successful, the function returns `kos::Ok`, otherwise it returns an error code.
`StartEntityWithUserHandlesEx()`	Purpose Starts a process a passes the list of handles to it. The names of the handles can be defined by the user. Parameters • [in] `runPath` – path to the executable file to be started. • [in] `info` – structure containing the `StartEntityInfo` process startup parameters. • [in] `userHandles` – list of custom handles that are passed to the process when it is started. • [out] `resInfo` – structure containing the `StartEntityResultInfo` process startup results. • [in] `containerPath` – path to the isolated data storage of a process. Returned values: If successful, the function returns `kos::Ok`, otherwise it returns an error code.
`RestartEntity()`	Purpose Restarts the process. Parameters • [in] `entId` – handle that identifies the running process. Returned values: If successful, the function returns `kos::Ok`, otherwise it returns an error code.
`GetEntityIds()`	Purpose Gets information about the processes executed by the ExecutionManager component. Parameters • [out] `entInfos` – list of handles identifying running processes. Returned values: If successful, the function returns `kos::Ok`, otherwise it returns an error code.

Function	Information about the function
`ShutdownEntity()`	Purpose Sends a termination signal to a process. Parameters • [in] `entId` – handle that identifies the running process. Returned values: If successful, the function returns `kos::Ok`, otherwise it returns an error code.
`StopEntity()`	Purpose Immediately stops the execution of a process. Parameters • [in] `entId` – handle that identifies the running process. Returned values: If successful, the function returns `kos::Ok`, otherwise it returns an error code.
`StartApplication()`	Purpose Starts process from the KPA package. Parameters • [in] `appName` – name of the KPA package. • [in] `info` – structure containing the `StartEntityInfo` process startup parameters. • [out] `resInfo` – structure containing the `StartEntityResultInfo` process startup results. Returned values: If successful, the function returns `kos::Ok`, otherwise it returns an error code.

Example of starting and stopping a process by using the `StartEntity()` and `StopEntity()` functions:

client.cpp

```
int main(int argc, const char *argv[])
{
    // ...
    const fs::path appPath{"/application"};

    execmgr::IApplicationController::StartEntityResultInfo result;
    execmgr::IApplicationController::StartEntityInfo info;

    info.entityName = std::string{"application.Application"};
    info.eiid = std::string{"application.Application"};
    info.args = std::vector<std::string>{"1", "ARG1", "ARG2" , "ARG3"};
    info.envs = std::vector<std::string>{"ENV1=10", "ENV2=envStr"};

    std::cout << "Starting application from elf\n";

    // Starting a process
    if (ac->StartEntity(appPath, info, result) != kos::Ok)
    {
        std::cerr << "Cannot start application from " << appPath << std::endl;
        return EXIT_FAILURE;
    }

    std::cout << "Application started with process sid " << result.sid << "\n";

    auto AppId = result.entId;

    // Stopping a process
    if (ac->StopEntity(AppId) != kos::Ok)
    {
        std::cerr << "Cannot stop process " << appPath << std::endl;
```

```
        return EXIT_FAILURE;
    }

   // ...
}
```

Example of starting a process and passing a custom handle to it by using the `StartEntityWithUserHandlesEx()` function:

client.cpp

```
int main(int argc, const char *argv[])
{
    // ...
    const fs::path appPath{"/application"};

    execmgr::IApplicationController::StartEntityResultInfo result;
    execmgr::IApplicationController::StartEntityInfo info;

    info.entityName = std::string{"application.Application"};
    info.eiid = std::string{"application.Application"};
    info.args = std::vector<std::string>{"1", "ARG1", "ARG2" , "ARG3"};
    info.envs = std::vector<std::string>{"ENV1=10", "ENV2=envStr"};

    std::cout << "Starting application from elf\n";

    // Starting a process and passing a custom handle to it
    if (ac->StartEntityWithUserHandlesEx(
            appPath,
            info,
            {
                {
                    .id = "vfs",
                    .handle = nk_handle_desc(file, OCAP_FILEOP_READ | OCAP_FILEOP_WRITE | OCAP_FILEOP_STAT |
OCAP_FILEOP_CLOSE | OCAP_HANDLE_TRANSFER),
                },
            },
            result) != kos::Ok)
    {
        std::cerr << "Can not start entity from " << appPath << std::endl;
        return EXIT_FAILURE;
    }

    // ...
}
```

application.cpp

```
int main(int argc, const char *argv[])
{
    std::cout << "application" << " started\n";

    // Verify that the custom handle was received
    auto h = KosTaskLookupResource("vfs");
    if (h == INVALID_HANDLE)
    {
        std::cout << "Cannot receive vfs handle" << std::endl;
        return EXIT_FAILURE;
    }

    // ...
}
```

IStateProvider interface

This API is defined in the header file `sysroot-*-kos/include/component/execution_manager/i_state_control.h` from the KasperskyOS SDK.

The API lets you receive information about a process that was started by using the `StartEntity()` function from the IApplicationController API (see IApplicationController interface), including information about the reason for its termination.

Information about API functions is provided in the table below.

To get information about the process, call the `GetApplicationState()` function. The `entityId` input parameter is used by this function to receive the `EntityId`-type value that identifies the running process. Using the `appState` output parameter, this function returns information about a process in the `AppContext` structure. `AppContext` structure is defined in the header file `sysroot-*-kos/include/component/execution_manager/types.h` from the KasperskyOS SDK.

```
struct AppContext
{
    // State of the process.
    AppState                                    state{};
    // Reason for stopping a process.
    alm::execution_manager::ExecutableExitStatus exitStatus{};
    // Return code for a process
    // that stopped on its own.
    alm::execution_manager::ExecutableExitCode   exitCode{};
    // Return code that specifies the reason for
    // an unsuccessful attempt to start a process.
    alm::execution_manager::ExecutableExitReason exitReason{};
};
```

The `state` enumeration is a component of the `AppContext` structure and describes the state of a process. `State` enumeration is defined in the header file `sysroot-*-kos/include/component/execution_manager/types.h` from the KasperskyOS SDK.

```
enum class AppState : uint32_t
{
    None = 0,
    // Process started.
    Started,
    // Process restarted.
    Restarted,
    // Process was stopped by using the StopEntity() function
    // from the IApplicationController API.
    Stopped,
    // Process was stopped by using the ShutdownEntity() function
    // from the IApplicationController API.
    Completed,
    // Process stopped unexpectedly
    // as a result of an unhandled exception.
    Failed,
};
```

The `exitStatus` enumeration is a component of the `AppContext` structure and describes the reason for stopping a process. The `ExecutableExitStatus` enumeration is defined in the header file `sysroot-*-kos/include/alm/execution_manager/execution_manager_types.h` from the KasperskyOS SDK.

```
enum class [[nodiscard]] ExecutableExitStatus : std::underlying_type_t<TaskExitStatus>
{
// Process has not yet been started by the ExecutionManager component.
    ExitUninitialized = TaskExitStatus::TaskExitUninitialized,
    // Process stopped unexpectedly as a result of an unhandled exception.
    ExitUnexpected    = TaskExitStatus::TaskExitUnexpected,
    // Process terminated on its own,
    // including after receiving a stop signal
    // via a call of the ShutdownEntity() function from the IApplicationController API.
    ExitNormal        = TaskExitStatus::TaskExitNormal,
    // Process stopped by external request, including via a call of
    // the StopEntity() function from the IApplicationController API.
    ExitTerminated    = TaskExitStatus::TaskExitTerminated,
};
```

The numeric `exitCode` parameter is a component of the `AppContext` structure and contains the return code for a process that stopped on its own, including after receiving a stop signal. The values for this return code are defined by the developer of the KasperskyOS-based solution.

The numeric `exitReason` parameter is a component of the `AppContext` structure and contains the return code that specifies the reason for an unsuccessful attempt to start the process, or `rcOk` if the process started successfully. Parameter is defined in the header file `sysroot-*-kos/include/rtl_cpp/retcode.h` from the KasperskyOS SDK. This file contains return codes that are common for the APIs of all solution components and their constituent parts (see Return codes).

i_state_control functions

Function	Information about the function
`GetApplicationState()`	Purpose Requests information about a process. Parameters • [in] `entityId` – value of the `EntityId` type that identifies the running process. • [out] `appState` – link to the `AppContext` structure containing information about the process. Returned values: If successful, the function returns `kos::Ok`, otherwise it returns an error code.

Usage example:

client.cpp

```
int main(int argc, const char *argv[])
{
    // ...

    execution_manager::AppContext appContext{};

    if (rc = m_sc->GetApplicationState(entityId, appContext), rc != kos::Ok)
    {
        KOS_LOGERROR("GetApplicationState: failure with entity {}: {}", entityName, RC_GET_CODE(rc));
        return rc;
    }

    // ...
}
```

ISystemController interface

The API is defined in the header file `sysroot-*-kos/include/component/execution_manager/i_system_control.h` from the KasperskyOS SDK.

The API lets you receive information about processes that were started by means of the ExecutionManager component, and it lets you shut down the system.

Information about API functions is provided in the table below.

Receiving information about processes

To request information about processes, call the `GetSystemInfo()` function. Using the `ProcessInfos` output parameter, this function returns information about processes as a vector of `ProcessInfo` structures.

```
struct ProcessInfo
{
    // Process name.
    std::string  name;
    // Process class name.
    std::string  eiid;
    // State of the process.
    ProcessState state;
};
```

The `state` enumeration is a component of the `ProcessInfo` structure and describes the state of a process.

```
enum class ProcessState : uint32_t
{
    // Service constant.
    None = 0,
    // Process created.
    Created,
    // Process initialized.
    Prepared,
    // Process started.
    Running,
    // Process stopped unexpectedly.
    Failed,
    // Process stopped, but data on the process is available
    // to the GetProcessState() function of the IProcessControl interface.
    Finished,
    // Process terminated and does not exist in the system.
    NotExisting,
};
```

Shutting down the system

To shut down the system, call the `StopAllEntities()` function. This function does not have any parameters.

i_system_control.h functions

Function	Information about the function
`StopAllEntities()`	Purpose Stops all running processes, then terminates the ExecutionManager process, and sends a device shutdown request to the kernel. Parameters N/A Returned values If successful, the function returns `kos::Ok`, otherwise it returns an error code.

Function	Information about the function
`GetSystemInfo()`	Purpose Receives information about processes that were started by means of the ExecutionManager component. Parameters • [out] `ProcessInfos` – vector of `ProcessInfo` structures containing information about processes. Returned values: If successful, the function returns `kos::Ok`, otherwise it returns an error code.

Usage example:

client.cpp

```
int main(int argc, const char *argv[])
{
    // ...

    alm::execution_manager::ProcessInfos infos{};
    res = sysCtl->GetSystemInfo(infos);
    if (res != kos::Ok)
    {
        std::cerr << fmt::format("[{}] Failed to get system info. res=0x{:x}\n", selfName, res);
        return EXIT_FAILURE;
    }

    if (sc->StopAllEntities() != kos::Ok)
    {
        std::cerr << "Cannot stop all processes\n";
        return EXIT_FAILURE;
    }

    // ...
}
```

PackageManager component

The API is defined in the header files located in the directory `sysroot-*-kos/include/component/package_manager/` from the KasperskyOS SDK.

The PackageManager component usage scenario is described in the article titled PackageManager component usage scenario.

The package_manager_proxy.h interface

This API is defined in the header file `sysroot-*-kos/include/component/package_manager/kos_ipc/package_manager_proxy.h` from the KasperskyOS SDK.

The interface contains function `CreatePackageManager()` for getting the pointer to the instance of the `IPackageManager` interface that is required for working with the PackageManager component. Using the `cfg` input parameter, this function accepts configuration parameters in the `PackageManagerConfig` structure. All fields of this structure are optional.

```
struct PackageManagerConfig
{
    // Name of the IPC channel for connecting to the PackageManager process.
    std::string const mainConnection = KOS_PKGMGR_CONN_MAIN;
    // Name of the endpoint that implements the IPackageController interface.
    std::string const pkgControlInterface = KOS_PKGMGR_IFACE_PKGC;
    // PackageManager process class name.
    std::string const pkgMgrServerName = KOS_PKGMGR_SERVER_NAME;
};
```

package_manager_proxy.h functions

Function	Information about the function
`CreatePackageManager()`	Purpose Gets the pointer to the instance of the `IPackageManager` interface that is required for working with the PackageManager component. Parameters • [in] `cfg` – `PackageManagerConfig` structure containing the configuration settings for the connection to the `PackageManager` process. • [out] `iPkgMgr` – pointer to the instance of the `IPackageManager` interface. Returned values: If successful, the function returns `kos::rtl::Ok`, otherwise it returns an error code.

Usage example:

client.cpp

```
#include <component/package_manager/kos_ipc/package_manager_proxy.h>

namespace pkgmgr = package_manager;

int main(int argc, const char *argv[])
{
    // ...

    pkgmgr::IPackageManagerPtr iPkgMgr;
    std::string mainConnection{"PkgMgrEntity"};
    std::string pkgControlInterface{"kl.package_manager.PackageManager.ipkgc"};
    pkgmgr::ipc::PackageManagerConfig cfg{mainConnection, pkgControlInterface};

    if (CreatePackageManager(cfg, iPkgMgr) != kos::rtl::Ok)
        return EXIT_FAILURE;

    // ...
}
```

IPackageManager interface

The API is defined in the header file `sysroot-*-kos/include/component/package_manager/i_package_manager.h` from the KasperskyOS SDK.

The `IPackageManager` interface lets you receive the pointer to the `IPackageController` interface. This interface is intended for installing KPA packages to a KasperskyOS-based solution, and gets information about these packages.

i_package_manager.h functions

Function	Information about the function
`GetPackageController()`	Purpose Gets the pointer to an instance of the `IPackageController` interface. Parameters • [out] `pkgController` – pointer to the instance of the `IPackageController` interface. Returned values: If successful, the function returns `kos::rtl::Ok`, otherwise it returns an error code.

Usage example:

client.cpp

```
int main(int argc, const char *argv[])
{
    // ...

    pkgmgr::IPackageControllerPtr pkgc;
    if (iPkgMgr->GetPackageController(pkgc) != kos::rtl::Ok)
        return EXIT_FAILURE;
    // ...
}
```

IPackageController interface

The API is defined in the header file `sysroot-*-kos/include/component/package_manager/i_package_control.h` from the KasperskyOS SDK.

The API lets you do the following:

- Install a KPA package in a KasperskyOS-based solution and remove the KPA package.
- Receive information about an installed KPA package: unique ID, data from the KPA package manifest, and the KPA package status (installed or removed).

Information about API functions is provided in the table below.

Installing a KPA package

To install a KPA package, call the `InstallPackage()` function. Using the `pkgInfo` input parameter, this function receives data to verify the certificates of an installed KPA package in the `InstallPackageInfo` structure. All fields of this structure are optional.

```
struct InstallPackageInfo {
    // Full name of the KPA package external signature file.
    std::filesystem::path signaturePath   = "";
    // Full name of the KPA package index file.
```

```
    std::filesystem::path indexPath       = "";
    // KPA package installation mode.
    InstallMode           installMode     = InstallMode::Update;
};
```

The `installMode` parameter determines the behavior of the `InstallPackage()` function when attempting to install a KPA package whose unique identifier is already in the system. The unique identifier is determined by the value of the `id` key of the `application` object. Parameter `installMode` can take the following values:

- `Update` – installation is allowed only if the version of the new KPA package is greater than the version of the already installed KPA package. The version of a KPA package is determined by the value of the `version` key of the `application` object in the KPA package manifest. If the versions are equal or the version of the new KPA package is lower, the installation will end with an error.
- `Force` – the KPA package will be installed regardless of the version comparison result. The previously installed KPA package will be removed, and then the new KPA package will be installed.

Deleting a KPA package

To remove a KPA package, call the `UninstallPackage()` function.

Getting information about installed KPA packages

To get the unique IDs of installed KPA packages, call the `ListInstalledPackages()` function. Using the `pkgUIDs` output parameter, the function returns a list of unique IDs of installed packages.

To get data on the KPA package manifest, call the `GetManifest()` function. Using the `manifest` output parameter, the function returns the pointer of the instance of the `IPackageManifest` interface that can be used to access the KPA package manifest key values. For more details, refer to IPackageManifest interface.

Notifications about the statuses of KPA packages in a KasperskyOS-based solution

The PackageManager component implements a mechanism for notifications about the statuses of KPA packages in a KasperskyOS-based solution, thereby enabling you to track changes to their statuses. To receive notifications, you must create a notification queue by using the `CreateEventQueue()` function.

To extract notifications from this queue, you must use the `GetEvents()` function. Using the `pkgEvents` output parameter, the function returns the status of KPA packages (since the moment when the queue was created) in the `PackageEvent` structure.

```
struct PackageEvent
{
    enum class Type
    {
        // KPA package was installed.
        INSTALL,
        // KPA package was removed.
        UNINSTALL,
        // Notification queue overflowed.
        OVERFLOWED,
    };
    // KPA package status.
```

```
    Type type;
    // Unique ID of the KPA package.
    std::string uid;
};
```

To delete the notification queue, call the `DestroyEventQueue()` function.

i_package_control.h functions

Function	Information about the function
`InstallPackage()`	Purpose Installs a KPA package. Parameters • [in] `pkgName` – name of the KPA package to be installed. • [out] `pkgUID` – unique ID of the installed KPA package. • [in] `pkgInfo` – `InstallPackageInfo` structure that contains data for verifying certificates of an installed KPA package. Returned values: If successful, the function returns `kos::rtl::Ok`, otherwise it returns an error code.
`UninstallPackage()`	Purpose Removes a KPA package. Parameters • [in] `pkgUID` – unique ID of the KPA package to be removed. Returned values: If successful, the function returns `kos::rtl::Ok`, otherwise it returns an error code.
`ListInstalledPackages()`	Purpose Gets the unique IDs of installed KPA packages. Parameters • [out] `pkgUIDs` – array of unique IDs of installed KPA packages. Returned values: If successful, the function returns `kos::rtl::Ok`, otherwise it returns an error code.
`GetPackageVersions()`	Purpose Gets the KPA package versions. Parameters • [in] `pkgID` – identifier of the KPA package for which the list of versions is returned. • [out] `versions` – list of KPA package versions. Returned values: If successful, the function returns `kos::rtl::Ok`, otherwise it returns an error code.

Function	Information about the function
`GetManifest()`	Purpose Gets the pointer to the instance of the `IPackageManifest` interface required for receiving data on the KPA package manifest. Parameters • [in] `pkgUID` – unique ID of the KPA package whose manifest data is required. • [out] `manifest` – pointer to the instance of the `IPackageManifest` interface. Returned values: If successful, the function returns `kos::rtl::Ok`, otherwise it returns an error code.
`GetArtifactBlob()`	Purpose Gets the contents of a KPA package component. Parameters • [in] `pkgUID` – unique ID of the KPA package containing the requested component. • [in] `artifactType` – type of KPA package component. For more details, refer to List of "components" objects. • [in] `artifactName` – name of the KPA package component. • [out] `blob` – array of bytes representing the contents of the KPA package component. Returned values: If successful, the function returns `kos::rtl::Ok`, otherwise it returns an error code.
`CreateEventQueue()`	Purpose Creates a queue for notifications about the statuses of KPA packages. Parameters • [out] `queueHandle` – link to the handle of the notification queue. Returned values: If successful, the function returns `kos::rtl::Ok`, otherwise it returns an error code.
`DestroyEventQueue()`	Purpose Deletes the queue for notifications about the statuses of KPA packages. Parameters • [in] `queueHandle` – handle of the notification queue. Returned values: If successful, the function returns `kos::rtl::Ok`, otherwise it returns an error code.
`GetEvents()`	Purpose Gets the statuses of KPA packages from the notification queue. Parameters • [in] `queueHandle` – handle of the notification queue. • [out] `pkgEvents` – `PackageEvent` structure containing the statuses of KPA packages. Returned values: If successful, the function returns `kos::rtl::Ok`, otherwise it returns an error code.

Function	Information about the function
`GetNotificationsHandle()`	Purpose Receives notifications about the installation or removal of a KPA package, but without the capability to specify which KPA package was modified. Obsolete function; you are advised to use the `GetEvents()` function. Parameters • [out] `notificationsHandle` – link to the handle that is used to process notifications. Returned values: If successful, the function returns `kos::rtl::Ok`, otherwise it returns an error code.

Usage example:

client.cpp

```
#include <component/package_manager/kos_ipc/package_manager_proxy.h>

namespace pkgmgr = package_manager;

int main(int argc, const char *argv[])
{
    // ...

    // Create a queue for notifications about the statuses of KPA packages.
    nk_handle_desc_t& eventQueue;
    if (pkgc->CreateEventQueue(eventQueue) != kos::rtl::Ok)
        return EXIT_FAILURE;

    // Install the KPA package named test1.
    std::string uid1;
    const std::string pkgName1{"test1"};
    const package_manager::InstallPackageInfo pkgInfo1{"", "", true};

    if (pkgc->InstallPackage(pkgName1, uid1, pkgInfo1) != kos::rtl::Ok)
        return EXIT_FAILURE;

    // Install the KPA package named test2.
    std::string uid2;
    const std::string pkgName2{"test2"};
    const package_manager::InstallPackageInfo pkgInfo2{"/test2.kcat", "/test2.kidx", true};
    if (pkgc->InstallPackage(pkgName2, uid2, pkgInfo2) != kos::rtl::Ok)
        return EXIT_FAILURE;

    // Get the pointer to the instance of the IPackageManifest interface
    // for the KPA package named test1.
    pkgmgr::IPackageManifestPtr manifest;
    if (pkgc->GetManifest(uid1, manifest) != kos::rtl::Ok)
        return EXIT_FAILURE;

    // Get the unique IDs of installed KPA packages.
    std::vector<std::string> pkgUIDs
    if (pkgc->ListInstalledPackages(pkgUIDs) != kos::rtl::Ok)
        return EXIT_FAILURE;

    // Remove the KPA package named test1.
    if (pkgc->UninstallPackage(uid1) != kos::rtl::Ok)
        return EXIT_FAILURE;

    // Remove the KPA package named test2.
    if (pkgc->UninstallPackage(uid2) != kos::rtl::Ok)
        return EXIT_FAILURE;

    // Get the statuses of KPA packages from the notification queue.
    std::vector<package_manager::PackageEvent> pkgEvents;
    if (pkgc->GetEvents(eventQueue, pkgEvents) != kos::rtl::Ok)
        return EXIT_FAILURE;

    // Delete the notification queue.
    if (pkgc->DestroyEventQueue(eventQueue) != kos::rtl::Ok)
        return EXIT_FAILURE;

    if (KnHandleClose(eventQueue) != rcOk)
        return EXIT_FAILURE;
```

```
    // ...
}
```

IPackageManifest interface

This API is defined in the header file `sysroot-*-kos/include/component/package_manager/i_package_manifest.h` from the KasperskyOS SDK.

The API lets you get the KPA package manifest key values. For more details about the available keys of the KPA package manifest, see "KPA package manifest"

Information about API functions is provided in the table below.

Getting program ID

To get the ID of a program that was installed from a KPA package, call the `GetPackageID()` function.

Getting information about KPA package components

To get information about KPA package components, call the `GetApplicationsInfo()` function. Using the `applications` output parameter, the function returns information about KPA package components as a vector of `ApplicationInfo` structures.

```
struct ApplicationInfo
{
    // Name of the KPA package component.
    std::string name;
    // Path of the directory relative to the path /<package name>/res
    // in which the KPA package component is installed.
    std::string path;
    // Unused parameter.
    std::string iconPath;
    // Unused parameter.
    std::string containerPath;
    // Unused parameter.
    std::string etcPath;
    // Checksum of the KPA package component file.
    std::string digest;
    // Object in JSON format.
    ManifestType extensions;
    // Type of KPA package component.
    std::string type;
};
```

An `extensions` structure element describes an object in JSON format. `ManifestType` is an alias of the `nlohmann:json` type.

Getting program startup configurations

To get information about the startup configurations of a program installed from a KPA package, call the `GetRunConfigurationsInfo()` function. Using the `runConfigurations` output parameter, the function returns information about program startup configurations as a vector of `RunConfigurationInfo` structures.

```
struct RunConfigurationInfo
{
    // Startup configuration ID that is unique within the specific KPA package.
    std::string id;
    // Startup configuration name.
    std::string name;
    // Program security class.
    std::string eiid;
    // Path to the KPA package component file.
    std::string path;
    // List of arguments in the form of a string array.
    std::vector<std::string> args{};
    // List of environment variables.
    std::vector<std::string> envs{};
    // Indicates whether the startup configuration is the primary one during program startup:
    // true means that it is primary, and false means that it is not primary.
    bool primary{false};
    // Indicates whether this configuration is started automatically:
    // true means that it is started automatically, and false means that it is not.
    bool autorun{false};
};
```

Receiving information about a program

To get information about a program installed from a KPA package, call the `GetPackageInfo()` function. Using the `packageInfo` output parameter, the function returns information about startup configurations in the `PackageInfo` structure.

```
struct PackageInfo
{
    // Unique ID of the program.
    std::string id;
    // Program name.
    std::string name;
    // Program version.
    std::string version;
    // Unused parameter.
    std::string title;
    // Unused parameter.
    std::string summary;
    // Program description.
    std::string description;
    // Program build number.
    std::string buildNumber;
    // Logical value indicating whether the program is a system program:
    // true means that it is, while false means that it is not.
    bool systemApplication;
    // Object of an arbitrary format.
    ManifestType extensions;
};
```

Getting information about the isolated storage of a program

To get information about the isolated storage of a program, call the `GetPrivateStorageInfo()` function. Using the `packageInfo` output parameter, the function returns information about startup configurations in the `PrivateStorage` structure.

```
struct PrivateStorage
{
    // Program data storage size in MB.
    size_t size;
    // Program data storage file system type.
    std::string fsType;
};
```

Getting information about objects of an arbitrary format

To get the KPA package manifest in JSON format, call the `Get()` function.

i_package_manifest.h functions

Function	Information about the function
`GetPackageID()`	Purpose Gets the unique ID of a program installed from a KPA package. Parameters • [out] `id` – unique ID of the program. Returned values: If successful, the function returns `kos::rtl::Ok`, otherwise it returns an error code.
`GetApplicationsInfo()`	Purpose Gets information about KPA package components. Parameters • [out] `applications` – vector of `ApplicationInfo` structures containing information about KPA package components. Returned values: If successful, the function returns `kos::rtl::Ok`, otherwise it returns an error code.
`GetRunConfigurationsInfo()`	Purpose Gets information about the startup configurations of a program installed from a KPA package. Parameters • [out] `runConfigurations` – vector of `RunConfigurationInfo` structures containing information about startup configurations. Returned values: If successful, the function returns `kos::rtl::Ok`, otherwise it returns an error code.

Function	Information about the function
`GetPackageInfo()`	Purpose Gets information about a program installed from a KPA package. Parameters • [out] `packageInfo` – `PackageInfo` structure containing information about the program. Returned values: If successful, the function returns `kos::rtl::Ok`, otherwise it returns an error code.
`GetPrivateStorageInfo()`	Purpose Gets information about the isolated storage of a program. Parameters • [out] `packageInfo` – `PrivateStorage` structure containing information about the isolated storage of a program. Returned values: If successful, the function returns `kos::rtl::Ok`, otherwise it returns an error code.
`Get()`	Purpose Gets the KPA package manifest in JSON format. Parameters N/A Returned values Returns the pointer of an object with the type `pkgmgr::IPackageManifest::ManifestType`.

Usage example:

client.cpp

```
#include <component/package_manager/kos_ipc/package_manager_proxy.h>

namespace pkgmgr = package_manager;

int main(int argc, const char *argv[])
{
    // ...

    // Get the pointer to the instance of the IPackageManifest interface
    // for the KPA package containing the unique ID uid1.
    pkgmgr::IPackageManifestPtr manifest;
    if (pkgc->GetManifest(uid1, manifest) != kos::rtl::Ok)
        return EXIT_FAILURE;

    // Get unique ID of the program.
    std::string packageID;
    if (manifest->GetPackageID(packageID) != kos::rtl::Ok)
        return EXIT_FAILURE;

    std::cout << packageID << '\n';

    // Get information about KPA package components.
    std::vector<package_manager::IPackageManifest::ApplicationInfo> infos;
    manifest->GetApplicationsInfo(infos);

    for (auto &info: infos)
    {
        std::cout << info.name << " " << info.path << "\n";
    }

    // Get the KPA package manifest in JSON format.
    pkgmgr::IPackageManifest::ManifestType jsonManifest = manifest->Get();

    std::cout << jsonManifest["package"]["description"] << '\n';
```

```
    // Get information about the startup configurations of a program installed from a KPA package.
    std::vector<package_manager::IPackageManifest::RunConfigurationInfo> runConfigurationInfos;
    manifest->GetRunConfigurationsInfo(runConfigurationInfos);

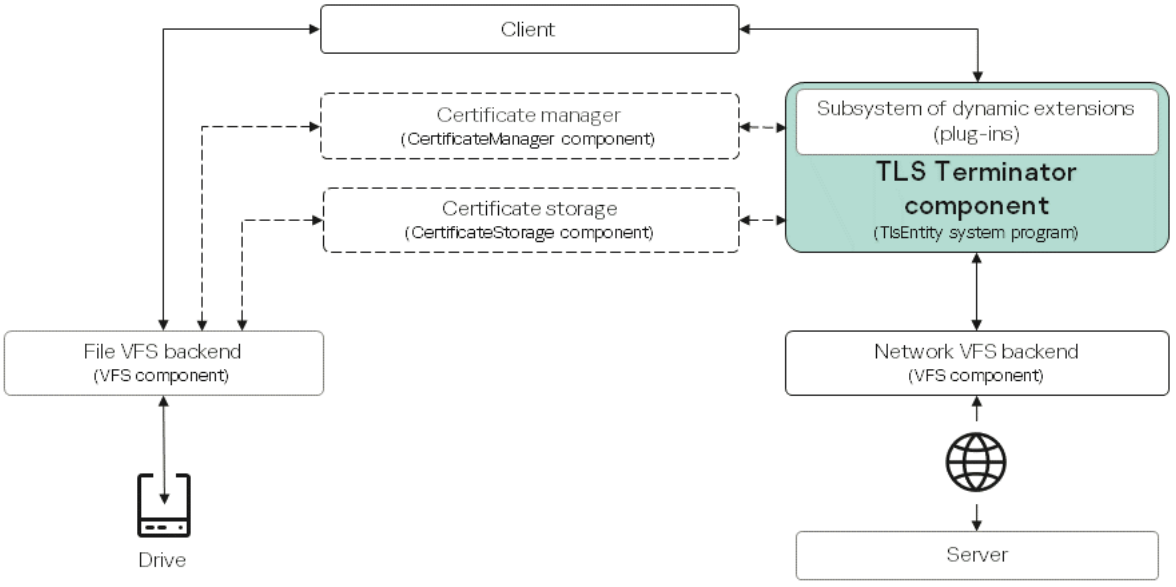
    for (auto &runConfigurationInfo : runConfigurationInfos)
    {
        std::cout << "RunConfiguration.id = " << runConfigurationInfo.id << ", autorun = " <<
runConfigurationInfo.autorun << "\n";
    }

    // ...
}
```

TLS Terminator component

The TLS Terminator component supplied as part of KasperskyOS Community Edition SDK ensures secure data transmission over the network between the client and the server using the Transport Layer Security (TLS) protocol.

The TLS Terminator component is built into the network interaction between the client and the network VFS backend. When a client uses network functions from the `libc` library, calls of these functions are redirected to the TLS Terminator component. The TLS Terminator component implements wrapper functions for standard calls of the network VFS backend. These wrappers ensure that TLS connections are established, outbound data is encrypted before it is sent to the network VFS backend, and inbound data is decrypted before being returned to the client. This implementation of the component ensures data protection during transmission over the network without the need to modify the client code.



How the TLS Terminator component works

Client

Calls functions from the `libc` library (see the table below for a complete list of supported functions) as if it were working directly with the network VFS backend.

TLS Terminator

Intercepts function calls from the client to the `libc` library and replaces them with wrapper functions implementing TLS. TLS connections are managed by using the Mbed TLS and OpenSSL libraries, which implement TLS protocols and cryptographic algorithms.

A subsystem of dynamic extensions (plug-ins) lets you extend the functionality of the TLS Terminator component without changing its source code by using dynamic libraries developed by the user.

Network VFS backend

It is provided by the VFS component that implements the network stack.

File VFS backend

It is provided by the VFS component that implements file systems.

Certificate manager

It is provided by the CertificateManager component, which lets you implement custom algorithms for verifying certificates received during a TLS connection. By default, the CertificateManager component implements an algorithm that rejects all certificate chains. Therefore, when this component is included in the solution, you must develop your own verification algorithm. The CertificateManager component may be absent, in which case the verification is performed using the resources of cryptographic libraries.

Certificate store

It is provided by the CertificateStorage component, which stores root certificates and intermediate chains. This component may be absent if certificates are stored directly in the file system.

API of the libc library supported by the TLS Terminator component

`accept()`	`send()`	`bind()*`	`getprotobyname()*`
`close()`	`sendto()`	`fcntl()*`	`getprotobynumber()*`
`connect()`	`sendvec()`	`freeaddrinfo()*`	`getsockname()*`
`getsockopt()`	`setsockopt()`	`freeifaddrs()*`	`ioctl()*`
`poll()`	`shutdown()`	`getaddrinfo()*`	`listen()*`
`read()`	`socket()`	`gethostbyname()*`	`pipe()*`
`readv()`	`write()`	`getifaddrs()*`	`socketpair()*`
`recv()`	`writev()`	`getnameinfo()*`	`subscribe()*`
`recvfrom()`		`getnetbyname()*`	`sysctl()*`
`recvvec()`		`getpeername()*`	

Functions marked with `*` in the table have standard POSIX behavior. The behavior of the rest of the functions is modified by the TLS Terminator component as indicated in the table below.

Modification of the behavior of libc functions by the TLS Terminator component

libc function	POSIX behavior of the function	Behavior of the function when using TLS Terminator
`accept()`	Accepts a connection request from the pending connection queue and creates a new socket handle for interaction	1. Accepts a connection request from the pending connection queue. 2. Performs a TLS handshake with the client. 3. Creates a new socket handle for interaction over a TLS connection.
`close()`	Closes the socket	1. Terminates the TLS connection. 2. Closes the socket.
`connect()`	Initiates a connection on the socket	1. Establishes a TCP connection with the server. 2. Performs a TLS handshake with the server. 3. Returns a secure socket handle.
`getsockopt()`	Gets socket options	1. Gets socket options. 2. Supports parameters specific for TLS Terminator, which are presented in the table below.
`read(), readv(), recv(), recvfrom(), recvvec()`	Receives data from the server	1. Receives encrypted data from the server. 2. Decrypts them via the TLS context. 3. Returns the decrypted data to the client.
`send(), sendto(), sendvec()`	Sends data to the server	1. Encrypts client data via the TLS context. 2. Sends encrypted data to the server over a TCP connection.
`setsockopt()`	Sets the parameters of a socket.	1. Sets the parameters of a socket. 2. Supports parameters specific for TLS Terminator, which are presented in the table below.
`shutdown()`	Terminates data transfer over the socket	1. Terminates the TLS connection. 2. Closes the TCP socket.
`socket()`	Creates a socket for network communication	1. Creates a socket. 2. Prepares a TLS context for subsequent use.
`poll()`	Tracks the state of file handles.	1. Monitors the state of TLS sockets. 2. Tracks the state of the TLS handshake and readiness for encryption/decryption.
`write(), writev()`	Sends data to the server	1. Encrypts outbound data using the TLS context. 2. Sends encrypted data to the server.

The `IPPROTO_TLS` value, which is not available in POSIX, has been implemented for the `level` parameter for the `getsockopt()` and `setsockopt()` functions. This value is defined in the `/opt/KasperskyOS-Community-Edition-<platform>-<version>/sysroot-*-kos/include/netinet/in.h` file from KasperskyOS Community Edition SDK. Syntax of functions:

```
int setsockopt(int socket, int level, int option_name, const void *option_value, socklen_t option_len);
int getsockopt(int socket, int level, int option_name, void *restrict option_value, socklen_t *restrict
option_len);
```

The `level` parameter specifies the level at which the `option_name` socket parameter name is defined. If the `level` parameter is equal to `IPPROTO_TLS` when calling the `getsockopt()` and `setsockopt()` functions, the TLS Terminator component processes the `option_name` socket parameter names that are available for this level from the table below.

Socket parameter names and their corresponding function behavior

Socket parameter name	Behavior of the setsockopt() function	Behavior of the getsockopt() function
`TLS_REMOTE_HOSTNAME`	Sets the name of the server to which the TLS connection is being established. This name will be used in the Server Name Indication (SNI) extension when performing the TLS handshake.	Gets the name of the server to which the TLS connection is being established.
`TLS_OCSP_STAPLING_CHECK`	Enables an OCSP Stapling check (confirmation that the certificate is valid) for the specified client socket.	This parameter is not processed.
`TLS_SESSION_ID`	This parameter is not processed.	Gets the ID of the TLS session.
`TLS_VERIFY_RESULT`	This parameter is not processed.	Returns the server certificate validation result as a combination of error flags.

If the default functionality of the TLS Terminator component is insufficient, it can be extended by using a plug-in mechanism. For more details, refer to Extending TLS Terminator functionality.

Contents of the TLS Terminator component

The component consists of the following files:

- Executable file `/opt/KasperskyOS-Community-Edition-<platform>-<version>/sysroot-*-kos/bin/tls`
- Formal specification file `/opt/KasperskyOS-Community-Edition-<platform>-<version>/sysroot-*-kos/include/kl/TlsEntity.edl` and the header file generated from it
- Solution security policy description file `/opt/KasperskyOS-Community-Edition-<platform>-<version>/sysroot-*-kos/include/kl/TlsEntity.psl`

Adding TLS Terminator to a KasperskyOS-based solution

To add the TLS Terminator component to a KasperskyOS-based solution, you must do the following:

- Add the following commands to the CMakeLists.txt root file:

```
# Connect the tls package for working with TLS Terminator.
find_package (tls REQUIRED COMPONENTS ENTITY CLIENT_LIB)
include_directories (${tls_INCLUDE})
```

- When building a solution (CMakeLists.txt file for the Einit program) using the `CMake` command `set_target_properties()`, in the `EXTRA_ARGS` and `EXTRA_ENV` properties, define the startup parameters and environment variables of the TLS Terminator component and assign the necessary values to them. For example:

```
# Define the startup parameters and environment variables for TLS Terminator.
set_target_properties (${tls_ENTITY} PROPERTIES
EXTRA_ARGS "
  - \"--cert\"
  - \"/ssl/client_crt_rsa_2048.pem\"
  - \"--key\"
  - \"/ssl/client_key_rsa_2048.pem\"
  - \"--server_min_rsa_keys_size\"
  - \"2048:2048:2048\""
EXTRA_ENV "
    VFS_FILESYSTEM_BACKEND: client:kl.VfsEntity
    VFS_NETWORK_BACKEND: client:kl.VfsEntity"
)
```

  Specify TLS Terminator as the network VFS backend for the client program. To do so, in the `CMake` command `set_target_properties()`, in the `EXTRA_ENV` property, define a value for the environment variable of the VFS program:

```
set_target_properties (client PROPERTIES
EXTRA_ENV "VFS_NETWORK_BACKEND: client:kl.TlsEntity"
)
```

  Alternatively, the values of startup parameters and environment variables can be defined not only in the `CMakeLists.txt` file for the `Einit` program but also in the solution initialization description file named `init.yaml`. In the example below, TLS Terminator is specified as the network VFS backend for the `WebServer` program:

```
- name: secure_login.WebServer
  env:
    VFS_NETWORK_BACKEND: client:kl.TlsEntity
```

- Include the TLS Terminator component EDL file in the solution security policy description:

```
use EDL kl.TlsEntity
```

TLS Terminator startup parameters and environment variables

Startup parameters and environment variables are defined in the CMakeLists.txt file for the Einit program using the CMake command `set_target_properties()` in the `EXTRA_ARGS` and `EXTRA_ENV` properties. For more details, please refer to the section titled Adding TLS Terminator to a KasperskyOS-based solution.

Startup parameters

- **{`-c`|`--cacert`}** `<FILE|DIR>`

  Path to the root certificates.

- `{-s|--cert}<FILE>`

  Path to the client certificate.

- `{-k|--key}<FILE>`

  Path to the client private key.

- `{-r|--crl}<FILE|DIR>`

  Path to the list of revoked certificates.

- `{-t|--timeout}<NUMBER>`

  Timeout for socket read operations during the TLS handshake (in milliseconds).

- `{-g|--log-level}<NUMBER>`

  Event log level (see the table below). TLS Terminator logs its own events and the events of crypto libraries.

  TLS Terminator log levels and their corresponding crypto library log levels

TLS Terminator log level	Crypto libraries log level	TLS Terminator log level description
`0 (NGLOG_SILENT)`	`0` (no logging)	No logging.
`1 (NGLOG_ERROR)`	`0` (no logging)	Errors are logged for TLS Terminator.
`2 (NGLOG_WARNING)` and `3 (NGLOG_INFO)`	`1` (error logging)	Errors, warnings, and informational messages are logged for TLS Terminator. Errors are logged for crypto libraries. Default value.
`4 (NGLOG_DEBUG)`	`2` (logging of errors and states) and `3` (logging of errors, states, and informational messages)	Full logging of TLS Terminator events. Errors, states, and informational messages are logged for crypto libraries.

- `{-e|--debug-if-file-exist}<FILE>`

  Enables the maximum log level when a file with the specified name exists. The availability of this file is checked for each new TLS connection. If the file does not exist, the log level defined via the `--log-level` parameter or the `NGLOG_ENV_LEVEL` environment variable is used.

- `{-d|--enable_dynamic_cert}`

  Automatic definition of the type of TLS authentication based on available certificates:

  - One-way TLS is activated when only the root certificate (`--cacert`) is available. The client verifies the server certificate but does not authenticate itself.

  - Two-way TLS is activated if a complete set consisting of a root certificate (`--cacert`), client certificate (`--cert`), and private key (`--key`) is available. In this mode, both sides of the TLS connection are mutually authenticated.

- `{-v|--cert_verify_required}`

  Enables mandatory verification of certificates by means of crypto libraries.

- `{-N|--server_cert_verify_disable}`

  Disable certificate verification using the resources of crypto libraries on the server side.

- `{-p|--plugin_path} <FILE>`

  Path to the dynamic library that implements the plug-in for extending the functionality of TLS Terminator. This parameter is used if the TLS Terminator component is built using a toolchain that supports dynamic linking (the `PLATFORM_SUPPORTS_DYNAMIC_LINKING` variable has the value of `true`).

- `{-o|--force_ocsp_stapling_check}`

  Enables server certificate validation using OCSP Stapling for all client connections. When supported by the OCSP Stapling server, the client receives a signed Certificate Authority Response (OCSP Response) containing information about the status of the certificate. The client checks the received certificate status. If the value is `good`, the connection is established. If it is `revoked` or `unknown`, it is declined.

- `--tls-min-version <VERSION>`

  Minimum version of the TLS protocol. Permissible values: `TLS-1.2`, `TLS-latest`.

- `--tls-max-version <VERSION>`

  Maximum version of the TLS protocol. Permissible values: `TLS-1.2`, `TLS-latest`.

- `--ciphersuite <NAME>`

  Adds a cipher suite to the list of supported ones.

- `--server_min_rsa_keys_size <ROOT:INTERMEDIATE:EE>`

  Minimum length of RSA keys for server certificates (in bits).

- `--client_min_rsa_keys_size <ROOT:INTERMEDIATE:EE>`

  Minimum length of RSA keys for client certificates (in bits).

Environment variables

- `NGLOG_LEVEL`

  Log level of the TLS Terminator component. Permissible values: from `0` to `4`. This environment variable is used if the startup parameter `--log-level` is not specified.

- `VFS_NETWORK_BACKEND=<VFS backend name>:<name of the IPC channel to the VFS process>`

  VFS backend for working with the network stack.

- `VFS_FILESYSTEM_BACKEND=<VFS backend name>:<name of the IPC channel to the VFS process>`

  VFS backend for working with file systems.

Extending TLS Terminator functionality

The plug-in mechanism extends the functionality of TLS Terminator so that you can implement custom logic into the operation of the TLS Terminator component without changing its source code. Plug-ins for the TLS Terminator component are dynamic libraries. To use the plug-in mechanism, you must do the following:

1. Develop a plug-in that implements the interface defined in the file `/opt/KasperskyOS-Community-Edition-<platform>-<version>/sysroot-*-kos/include/tls/tls_plugin_interface.h`. Implementation of the following interface functions is required:

   - `plugin_init()` – initialize the plug-in.
   - `plugin_fini()` – terminate the plug-in.
   - `plugin_get_version()` – get the plug-in version.

   Implementation of the rest of the interface functions is optional and depends on the tasks for which the plug-in is being developed.

2. When starting TLS Terminator, you must pass the path of the developed plug-in in the `--plugin_path` startup parameter. For more details, refer to TLS Terminator startup parameters and environment variables.

Plug-in mechanism details:

1. The plug-in is loaded into memory using the POSIX function `dlopen()`.
2. Version compatibility is verified by using the `plugin_get_version()` function, and the plug-in is initialized via the `plugin_init()` call.
3. When calling a function that can be intercepted by the TLS Terminator component:

   1. The component uses the POSIX function `dlsym()` to check whether the plug-in has an implementation of the called function.
   2. If the function is implemented in the plug-in, the component executes the plug-in version of the function. Otherwise, the TLS Terminator component executes its own version of the function.

   During initialization, the TLS Terminator component passes to the plug-in a set of pointers to its own implementations of functions through the `tls_functions` structure. This allows the plug-in to call these implementations as needed.

4. When shutting down, the TLS Terminator component terminates the plug-in by calling the `plugin_fini()` function.
5. The plugin is unloaded from memory using the POSIX function `dlclose ()`.

Toybox component

The Toybox component is an implementation of the toybox command line tool adapted to work in KasperskyOS.

Contents of the Toybox component:

- Executable file `/opt/KasperskyOS-Community-Edition-<platform>-<version>/sysroot-*-kos/bin/terminal_uart_entity`
- Formal specification files named `Terminal.edl` and `Terminal.idl` and the header files generated from them, `Terminal.edl.h` and `Terminal.idl.h`, which are located in the directory `/opt/KasperskyOS-Community-Edition-<platform>-<version>/sysroot-*-kos/include/kl`
- Static library `/opt/KasperskyOS-Community-Edition-<platform>-<version>/sysroot-*-kos/lib/libtoybox.a`.
- Header file `/opt/KasperskyOS-Community-Edition-<platform>-<version>/sysroot-*-kos/include/toybox/toybox.h`.

> Important! The Toybox component is experimental and subject to limitations described below. The component is available to users for testing purposes and for providing feedback.

The functionality of the Toybox component in KasperskyOS has the following differences from the implementation of the toybox tool in Linux due to the architectural features of KasperskyOS and its security policies:

- A limited set of commands is supported (the list is provided in the table below).
- Some commands have limited functionality compared to the original version (marked with an asterisk `*` in the table below).
- Some commands have their own implementation that replaces the original version (marked with two asterisks `**` in the table below).

Commands supported by the Toybox component

[	df*	file *	jobs**	mountpoint *	reboot**	strings	uptime *
base64 *	dirname	find *	kill**	mv *	reset	tail *	wc
basename	dmesg**	fmt	link	nl	rm	test	xxd
cat *	du *	free**	ln *	od	rmdir	top**	yes
chmod	echo	getopt *	ls *	patch *	sed *	touch *	zcat
clear	egrep	grep *	lsblk**	ping**	seq	true	
cmp	env	gunzip *	lspcie**	printf	sleep	truncate	
comm	exec**	head	lsusb**	pwd *	sort *	uname *	
cp *	false	help	md5sum *	readelf *	split	uniq	
date *	fgrep	hexedit	mkdir	readlink	stat *	unlink	

Commands with limited functionality in the Toybox component

Command	Unsupported command options and other limitations
base64	[-i]
cat	[-etuv]
cp	[-adFHLlnRsv]

Command	Unsupported command options and other limitations
`date`	`[-s]` The value of the TZ environment variable is ignored.
`df`	`[-aHhikP] [-t TYPE]`
`du`	`[-abcHKkLlmsx] [-d N]`
`file`	`[-bs]`
`find`	Unsupported search filters: • `-user UNAME` • `-nouser` • `-group GROUP` • `-nogroup` • `-prune` • `-xdev` • `-links N` • `-empty` • `-atime N[u]` • `-ctime N[u]` • `-mtime N[u]` • `-inum N` • `-context PATTERN` • `-newer FILE` • `-executable` • `-readable` • `-depth` • `-mindepth N` Search filters were added: • `-lname PATTERN` • `-ilname` Unsupported actions: • `-print0` • `-exec` • `-execdir` • `-ok` • `-okdir` • `-delete`
`getopt`	`[-a]`
`grep`	`[-ft] [-MS PATTERN]`

Command	Unsupported command options and other limitations
gunzip	[-ft]
ln	[-v]
ls	[-HLghns] [--color[=auto]]
md5sum	[-b] Hashes are not calculated based on SHA algorithms.
mountpoint	[-qdx] Operations with devices are not supported.
mv	[-FTvx] [-t TARGET]
patch	[-Rlsuv] [-d DIR] [-i FILE] [-p DEPTH] [-F FUZZ]
pwd	[-L]
readelf	[-A]
sed	[-i]
sort	[-bdiMsVz]
stat	[-tfL]
tail	[-fFs]
touch	[-h]
uname	[-m]
uptime	[-ps] Data on the number of active users and the average system load are not provided.

Commands with their own custom implementation in the Toybox component

Command	Command details
dmesg	Purpose Displays diagnostic messages from the KasperskyOS kernel generated when the system is loaded and running. Syntax `dmesg` Options N/A

Command	Command details
`exec`	Purpose Starts a process from an executable file using the `StartEntity()` function of the IApplicationController interface of the ExecutionManager component. Syntax `exec [--bg] [--eiid <className>] [--name <taskName>] <fileName> [--<arguments>…]` Options **`--bg`** Starts a process in the background. After starting a process in the background, the terminal will display the ID of the running process enclosed in square brackets. **`--eiid <className>`** Process class name. Unless otherwise specified, the name `exec_app.ExecApp` will be used. **`--name <taskName>`** Process name. If not specified, the process class name is used. **`fileName`** Executable file name. **`--<arguments>`** Command line arguments to be passed to the started process.
`free`	Purpose Displays data on physical memory based on the values of performance counters: • Installed physical memory (`total`) • Used memory, calculated as the difference between installed memory and free memory, shared memory and memory with the "execution access" attribute (`used = total - free - shmem - exsize`) • Free physical memory (`free`) • Physical memory used as shared memory (`shmem`) • Physical memory with the "execution access" attribute (`exsize`) Syntax `free [-bkmg]` Options **`-bkmg`** Display values in bytes/kilobytes/megabytes/gigabytes.

Command	Command details

`jobs`	Purpose
	Displays data on background processes obtained using the `GetApplicationState()` function of the IStateProvider interface of the ExecutionManager component in the following format:
	`[<jobNumber>] <taskName>: <state> [ExitCode=<exitCode>]`
	Where:
	• `jobNumber` – process ID
	• `taskName` – process name
	• `state` – process state
	• `exitCode` – return code for the process if the process stopped on its own (including after receiving a signal to stop)
	For example:
	`[1] DemoSuccess.3: Completed ExitCode=0`
	Syntax
	`jobs`
	Options
	N/A

`kill`	Purpose Stops background processes that were previously started by the `exec` command using the `StopEntity()` function of the IApplicationController interface of the ExecutionManager component. Syntax **`kill jobNumber1 [jobNumber2...]`** Options **`jobNumber<N>`** Process ID that can be obtained by using the `jobs` or `exec` commands (if `exec` was called with the `--bg` option).

Command	Command details

`lsblk`	Purpose
	Displays a list of detected block devices in the following format:
	`<idx> <devName> hw:<hwName> <{present|absent}>, <{removable|non-removable}> [<blockCount> * <blockSize> bytes] <flags>`
	Where:
	• `idx` – device index in the list
	• `devName` – device name
	• `hwName` – name of the driver that facilitates operation of the device
	• `present|absent` – device state: `present` (available) or `absent` (not available)
	• `removable|non-removable` – device type: `removable` or `non-removable`
	• `[<blockCount> * <blockSize> bytes]` – device size in bytes, defined as the result of multiplying `blockCount` (the number of blocks on the device) by `blockSize`
	• `flags` – additional device attributes: `raw` (raw device), `ro` (read-only), `rw` (read and write), `(partition)` (disk partition)
	For example:
	`1 SDv2 hw:pl181-sd present, non-removable [522240 * 512 bytes] rw (partition)`
	Syntax
	`lsblk`
	Options
	N/A

`lspcie`	Not supported on OrangePi RV2.
	Purpose
	Displays data about PCIe devices detected on the hardware platform. For each PCIe device, the following is indicated:
	• Device address in the following format:
	`<segment>:<bus>:<device>.<function>`
	  • `segment` – device segment number in hexadecimal format
	  • `bus` – PCIe bus number in hexadecimal format
	  • `device` – device number on the bus in hexadecimal format
	  • `function` – device function number in hexadecimal format
	• Vendor and device identifiers in the following format:
	`vendorId: <vID>, deviceId: <dID>`
	  • `vID` – vendor ID in hexadecimal format
	  • `dID` – device ID in hexadecimal format
	• Device class and its subclasses in the following format:
	`class: <class>, subclass: <subc>, interface: <intr>, revision: <rev>`
	  • `class` – device class code in hexadecimal format
	  • `subc` – device subclass code in hexadecimal format
	  • `intr` – software interface code in hexadecimal format
	  • `rev` – device revision number in hexadecimal format
	• Interrupts in the following format:

Command	Command details

	`IRQ: <irq>, pin: <pin>, mode: <mode>`
	• `irq` – interrupt line number
	• `pin` – interrupt pin number
	• `mode` – interrupt mode in hexadecimal format
	• List of base address registers (BAR) in the following format:
	`BAR<N>: <type> <{prefetchable|non-prefetchable}> @ <start> - <end>`
	• `N` – BAR number
	• `type` – BAR type: `IO` (I/O port), `MEM32` (32-bit memory), `MEM64` (64-bit memory)
	• `prefetchable|non-prefetchable` – BAR properties: `prefetchable` (supports preloading data into the cache), `non-prefetchable` (does not support preloading data into the cache)
	• `start` – BAR start address
	• `end` – BAR end address
	Syntax
	`lspcie`
	Options
	N/A

`lsusb`	Not supported in QEMU. Purpose Displays a list of USB devices detected on the hardware platform in the following format: `<N> <devName>: <bus>/<addr>/<bus_idx> <<vendor> <product>>` **`cfg`**`<cfg_idx>` **`(cnt=`**`<cfgN>`**`)`** Where: • `N` – serial number of the device in the list • `devName` – device name • `bus` – USB bus number • `addr` – device address on the USB bus • `idx` – device index • `vendor` – name of the device manufacturer • `product` – product name • `cfg_idx` – device configuration index • `cfgN` – device configuration number Syntax **`lsusb`** Options N/A

Command	Command details
`ping`	Purpose Checks network connections via ICMP requests. Syntax `ping [-c <count>] host` Options **`-c <count>`** Number of sent packets. **`host`** Target network host specified by IP address or domain name.
`reboot`	Purpose Restarts the hardware platform using the KnHalResetSystem() function. Syntax `reboot` Options N/A
`top`	Purpose Displays data on processes: • `PID` – process ID. • `PPID` – ID of the parent process. • `PR` – process priority. • `S` – process state: `R` (running), `S` (blocked) or `T` (terminated but not deleted), `U` (not initialized). • `THR` – number of threads in the process. • `VMEM` – size of process virtual memory, in memory pages. • `PMEM` – size of physical memory used by the process, in memory pages. • `VSHR` – size of process virtual memory mapped to shared physical memory, in memory pages. • `TIME` – total uptime of all processors used for execution of the defined process (in nanoseconds). • `TIME` – total uptime of all processors used for execution of the defined process in user mode (in nanoseconds). • `NAME` – process name. • `PATH` – name of the executable file in ROMFS from which the process was started. Syntax `top [-h] [-a] [-f <format>]` Options **`-h`** Help text. **`-a`** Additional information about processes (PR and UTIME). **`-f <format>`** Output format. Detailed information about the output format is specified in the Help text (option `-h`).

Adding Toybox to a KasperskyOS-based solution

> The Toybox component requires the ExecutionManager component. For details on how to work with the ExecutionManager component in a KasperskyOS-based solution, see the section titled Starting processes using the system program ExecutionManager.

To add the Toybox component to a KasperskyOS-based solution, you must do the following:

- Add the following commands to the CMakeLists.txt root file:

```
# Connect the uart package with the UART driver.
find_package (uart REQUIRED)

# Connect the Terminal package that facilitates interaction
# between the UART serial port and the virtual file system
# to implement standard input/output streams and line discipline.
find_package (Terminal REQUIRED COMPONENTS UART_ENTITY)

# Connect the toybox package.
find_package (toybox REQUIRED)
add_subdirectory (toybox)

# Create a CMake target to generate an .edl.h file for the EDL description of the class of processes
# started when the exec command of the Toybox component is executed.
nk_build_edl_files (exec_app_edl_files
                    NK_MODULE "exec_app"
                    EDL       "${NK_RESOURCES}/exec_app/ExecApp.edl")
```

- Pack the Toybox program into a KPA package and run it using the API of the PackageManager system program.

  For more details, refer to the uart_toybox example in the SDK.

- Include the EDL files in the solution security policy description:

```
use EDL toybox.Toybox
use EDL kl.Terminal
use EDL exec_app.ExecApp
```

> For more details, see the `uart_toybox` example located in `examples` included in KasperskyOS Community Edition SDK.

Kaspersky Control Flow Monitor component

The Kaspersky Control Flow Monitor (KCFM) component is designed to monitor and control the performance of processes by monitoring the control flow.

The KCFM component lets you do the following in a KasperskyOS-based solution:

- Detect anomalies in the behavior of the monitored process.
- If an anomaly is detected, restart the monitored process.
- [optional] Automatically generate the part of the solution security policy for program IPC calls to the core endpoints and the VFS system program.

> For more details, refer to the kcfm_echo_server example in the SDK.

How the KCFM component works

By using the source code of a program that needs to be monitored, you can build a control flow graph (CFG) for the program. This graph defines the complete set of system calls and calls to library functions that can be executed while the program is running, and all possible sequences of their execution. The CFG is built during program build.

While a process (program) is running, IPC calls made by the process are monitored. If program behavior is unexpected (anomalous), the KCFM component lets you stop and restart the monitored program.

> Important! The KCFM component is experimental and subject to limitations described below. The component is available to users for testing purposes and for providing feedback.

Limitations in the experimental version of the KCFM component

The KCFM component has the following limitations:

- The component is not included in the SDK for the OrangePi RV2 hardware platform.
- Only one process can be monitored in a KasperskyOS-based solution.
- Only programs written in C and C++ are supported.
- When using computed pointers to call functions in the program code, the tools for building a control flow graph cannot explicitly determine the function being called. In this case, an assumption is made about possible function calls, a signature with these calls is built, and a warning is issued.
- A call to functions for using hardware interrupt handlers in the program code is ignored and an appropriate warning is issued.

- When using explicitly linked API functions for dynamic libraries, a warning is issued that a call to these functions cannot be processed.
- If anomalies are detected in a running process, the KCFM component restarts the process. A restart is the only action available.

Contents of the KCFM component

The component consists of the following files:

- Executable files of modules for building a control flow graph of a monitored program, monitoring calls for a running program, and detecting anomalies based on the control flow graph. These files are located in `/opt/KasperskyOS-Community-Edition-<platform>-<version>/toolchain/bin` and `/opt/KasperskyOS-Community-Edition-<platform>-<version>/sysroot-*-kos/bin`.
- Formal specification files located in `/opt/KasperskyOS-Community-Edition-<platform>-<version>/sysroot-*-kos/include/kcfm`.
- `CMake` libraries for adding KCFM component modules to the solution located in `/opt/KasperskyOS-Community-Edition-<platform>-<version>/toolchain/lib/cmake/kcfm_tool`.
- Examples of solutions for demonstrating use of the KCFM component in `/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples/`.

Building a program control flow graph

The control flow graph is built during application build if the CMake command kcfm_generate_signature() has been added to the CMakeLists.txt file for building the program. The file created by the `kcfm_generate_signature()` command must be added to the KPA package of the application.

To track IPC calls made by a monitored program, you must include a specialized audit profile in the policy description. To do so, add the `CMake` command `kcfm_generate_audit_policy()` to the CMakeLists.txt file for building the monitored program.

The monitored program code can use static or dynamic libraries. In this case, these libraries must also be built via CMake, and the CMake commands `kcfm_make_graph_lib()` for building the call flow graphs of these libraries must be added to the build script.

Anomaly detection

Anomaly detection is performed by the `FlowChecker` module from the KCFM component. To add the `FlowChecker` module to a solution, you need to package the `FlowChecker` application into a KPA package and add it to the solution. Packing an application into a KPA package and adding the component to the solution is performed in the CMakeLists.txt build scripts. For more details, refer to the kcfm_echo_server example in the SDK.

Anomaly processing

A detected anomaly is processed by the `FlowMonitor` module from the KCFM component. To add the `FlowMonitor` module to a solution, you need to package the `FlowMonitor` application into a KPA package and add it to the solution. Packing an application into a KPA package and adding the component to the solution is performed in the CMakeLists.txt build scripts. For more details, refer to the kcfm_echo_server example in the SDK.

The `FlowMonitor` module performs the following actions:

- Starts the program using `ExecutionManager`.
- Monitors the status (started/stopped) of the program via `ExecutionManager`.
- If the program stops, it restarts the program using `ExecutionManager`.
- Receives and processes events for detecting anomalies in the behavior of the monitored process from the `FlowChecker` module. In this case, the `FlowMonitor` module performs the following actions:
  - Issue a command to stop the program.
  - Wait until the program terminates.
  - Issue a command to start the program.

Security policy generation

The KCFM component also lets you automatically generate part of the solution security policy in the form of a PSL file based on the program source code. The PSL file will list and allow all IPC calls to core endpoints and VFS system program interfaces that are used in the program code. The security policy is formed so as to allow the program to access the IPC calls in use and deny access to all of the unused ones.

To generate the PSL file, add the CMake command `kcfm_generate_audit_policy()` to the CMakeLists.txt file for building the program. The generated PSL file must be included in the solution security policy description.

Building a KasperskyOS-based solution

This section contains the following information:

- General solution build scenario
- Structure of a solution CMake project
- Typical contents of `CMakeLists.txt` files, init description template, and solution security policy description template in a solution CMake project
- Reference information on non-standard CMake commands specific to the solution build system
- Information on how to use dynamic libraries in solution

An overview of the solution building process

The solution consists of system software (including the KasperskyOS kernel, the Kaspersky Security Module, and system programs) and application software. *Application software* is designed for interaction with a solution user and for performing user tasks. Applications software is not included in KasperskyOS Community Edition SDK. *System programs* create the infrastructure for application software. For example, they facilitate hardware operations, support the IPC mechanism, and implement file systems and network protocols. A set of system programs (including drivers) is provided as executable files in the KasperskyOS Community Edition SDK. Transport libraries and transport code for using system program endpoints and system program components in the form of libraries are also provided. If necessary, you can create your own system programs, including those based on the provided libraries. An initializing program and a Kaspersky Security Module must be created for each specific solution. The build system uses source code generators to create them. The KasperskyOS kernel image is provided as part of the KasperskyOS Community Edition SDK.

To build a solution, you must complete the following steps:

1. Create formal specifications of the solution components for all programs except for the provided system programs.

   Formal specifications of the solution components for the provided system programs are included in the KasperskyOS Community Edition SDK. The KasperskyOS Community Edition SDK also contains the files of formal specifications of solution components for system program components provided as libraries. Therefore, for a system program that is created based on the provided libraries, it is not necessary to create all the files of the formal specification of a solution component because some of the files corresponding to the utilized libraries are provided in the KasperskyOS Community Edition SDK.

   EDL descriptions can be generated by using the `generate_edl_file()` CMake command.

2. Create a solution security policy description.

3. Generate a transport code.

   To complete this step, you need to use the CMake commands `nk_build_*dl_files()` and/or `add_nk_*dl()`. The input data for creating transport code are the formal specifications of the solution components. The transport code must be generated for programs that provide endpoints and/or use endpoints of other programs, unless the required transport libraries and/or transport code are provided as part of the KasperskyOS Community Edition SDK. Transport code does not need to be generated for the `Einit` initializing program.

4. Build executable files and libraries.

   To complete this step, you need to use standard CMake commands. If the program provides endpoints and/or uses the endpoints of other programs, the source code must include the transport code and/or link the executable file to the transport libraries.

5. Pack applications into KPA packages by using the CMake library kpa.

6. Create the initial_install.json and system_services.json configuration files for the ServiceManager system program. For more details, refer to ServiceManager system program.

7. Build the additional image (VendorROMFS). For more details, refer to Architecture and startup of a KasperskyOS-based solution.

8. Build the root file system rootfs, which includes the following:

   - Contents of the archive containing the libraries, files, and KPA packages necessary for the operation of system programs from the InitROMFS boot image.
   - Files and libraries required for the operation of system programs from the VendorROMFS image.
   - KPA packages of all applications included in the solution.
   - Configuration files for the ServiceManager system program, which is responsible for installing and running KPA packages.

   To complete this step, use the CMake library build_vendor_rom(). For usage examples, see the ./einit/CMakeLists.txt build scripts in the SDK examples.

9. Create a KasperskyOS-based boot image by using the CMake command generate_qemu_target() to create an image to run in QEMU or build_sd_image() to create an image to run on the hardware platform.

   For usage examples, see the ./einit/CMakeLists.txt build scripts in the SDK examples.

Using CMake from the contents of KasperskyOS Community Edition SDK

To automate the process of preparing the solution image, you need to configure the `CMake` build system. You can base this system on the build system parameters used in the examples from KasperskyOS Community Edition SDK.

`CMakeLists.txt` files use the standard `CMake` syntax, and commands and macros from libraries provided in KasperskyOS Community Edition SDK.

> When building with `CMake`, the flags `-Wall -Wextra -Wconversion -Wsign-conversion -Wformat=2 -Wformat-security -Werror=format-security -fPIE` are used by default.

Recommended structure of project directories

When creating a KasperskyOS-based solution, it is recommended to use the following directory structure in a project:

- In the project root, create a CMakeLists.txt root file containing the general build instructions for the entire solution.
- The source code of each program being developed should be placed into a separate directory within the `src` subdirectory.
- Create CMakeLists.txt files for building each application in the corresponding directories.
- Create a separate `einit` directory containing the `src` subdirectory where the solution security policy description security.psl should be placed.

  You can also use this directory to store any other files that need to be included in the solution image, such as the initial_install.json and system_services.json configuration files for the ServiceManager system program.
- Create a CMakeLists.txt file for building the solution image.
- The files of EDL, CDL and IDL descriptions should be put into the `resources/xdl/` directory in the project root. (An EDL description can be generated during the build by using the `generate_edl_file()` command.)

Example structure of project directories

```
example$ tree
.
├── CMakeLists.txt
├── hello
│   ├── CMakeLists.txt
│   ├── src
│   │   ├── hello.c
├── einit
│   ├── CMakeLists.txt
│   ├── src
│   │   ├── initial_install.json
│   │   ├── security.psl
│   │   ├── system_services.json
├── resources
│   ├──xdl
│       ├── Hello.idl
│       ├── Hello.cdl
│       ├── Hello.edl
```

Building a solution image

To build a solution image, you must use the `cmake` tool (the `toolchain/bin/cmake` executable file from KasperskyOS Community Edition SDK).

Build example:

```
# You can get information about the cmake tool run parameters
# via the cmake --help command, and from
# the official CMake documentation.

# Before building, you must set the values of the environment variables.
```

```
$ source /opt/KasperskyOS-Community-Edition-<platform>-<version>/common/set_env.sh

# Initialize the build system
$ cmake -B build -D CMAKE_TOOLCHAIN_FILE="$KOSCEDIR/toolchain/share/toolchain-*-kos.cmake"

# Build
# To build a solution image for QEMU and start QEMU with this image, you must
# specify the sim target.
$ cmake --build build --target sim
```

CMakeLists.txt root file

The `CMakeLists.txt` root file contains general build instructions for the entire solution.

The `CMakeLists.txt` root file must contain the following commands:

- `cmake_minimum_required (VERSION 3.29)` indicates the minimum supported version of `CMake`.

  For a KasperskyOS-based solution build, `CMake` version 3.29 or later is required.

  The required version of `CMake` is provided in KasperskyOS Community Edition SDK and is used by default.

- `include (platform)` connects the `platform` library of `CMake`.

- `include (platform/nk)` connects the `CMake` library for working with the NK compiler.

- `include (platform/options)` connects the `CMake` library for working with build options.

- `find_package (minimal_kos REQUIRED)` connects the `CMake` library for building the base boot image of KasperskyOS.

- `initialize_platform()` initializes the `platform` library.

- `project_header_default("STANDARD_GNU_17:YES" "STRICT_WARNINGS:NO")` sets the flags of the compiler and linker.

- **[Optional]** Connect and configure packages for the provided system programs and drivers that need to be included in the solution:

  - A package is connected by using the `find_package()` command.

  - After connecting a package, you must add the package-related directories to the list of search directories by using the `include_directories()` command.

  - For some packages, you must also set the values of properties by using the `set_target_properties()` command.

> `CMake` descriptions of system programs and drivers included in KasperskyOS Community Edition SDK, as well as their exportable variables and properties are located in the corresponding files at `/opt/KasperskyOS-Community-Edition-<platform>-<version>/sysroot-*-kos/lib/cmake/<program name>/<program name>Config.cmake`

- The solution image must be built using the `add_subdirectory (einit)` command.
- All applications to be built must be added by using the `add_subdirectory(<program directory name>)` command.

Example CMakeLists.txt root file

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.29)
project (example)

# Initializes the CMake library for the KasperskyOS SDK.
include (platform)
initialize_platform ()

# Set the linker and compiler flags.
project_header_default ("STANDARD_GNU_17:YES" "STRICT_WARNINGS:NO")

# Add package importing components for working with Virtual File System.
# Components are imported from the following file: /opt/KasperskyOS-Community-Edition-<platform>-
<version>/sysroot-*-kos/lib/cmake/vfs/vfsConfig.cmake
find_package (vfs REQUIRED COMPONENTS ENTITY CLIENT_LIB)
include_directories (${vfs_INCLUDE})

# Build the Hello application.
add_subdirectory (hello)

# Building a solution image
add_subdirectory (einit)
```

CMakeLists.txt files for building applications

The `CMakeLists.txt` file for building an application must contain the following commands:

- An EDL description of a process class for a program can be generated by using the `generate_edl_file()` command.
- If the program provides endpoints using an IPC mechanism, the following transport code must be generated:

  a. idl.h files are generated by the `nk_build_idl_files()` command

  b. cdl.h files are generated by the `nk_build_cdl_files()` command

  c. edl.h files are generated by the `nk_build_edl_files()` command

- `add_executable (<program name> "<path to the file containing the program source code>")` adds the program build target.
- `add_dependencies (<program name> <name of the edl.h file build target>)` adds a program build dependency on edl.h file generation.
- `target_link_libraries (<program name> <list of libraries>)` determines the libraries that need to be linked with the program during the build.

  For example, if the program uses file I/O or network I/O, it must be linked with the `vfs::client` transport library.

> `CMake` descriptions of system programs and drivers provided in the KasperskyOS Community Edition SDK, and descriptions of their exported variables and properties are located in the corresponding files at `/opt/KasperskyOS-Community-Edition-<platform>-<version>/sysroot-*-kos/lib/cmake/<program name>/<program name>Config.cmake`

- Commands of the CMake library kpa for packing an application into a KPA package.

Example CMakeLists.txt file for building a simple application

CMakeLists.txt

```
include (platform/kpa)

# Define the program name.
set (PROGRAM_NAME  Hello)
set (PACKAGE       ${PROGRAM_NAME}.kpa)

# Add the edl.h file build target.
nk_build_edl_files (${PROGRAM_NAME}_edl_files
                    NK_MODULE "hello"
                    EDL       "${NK_RESOURCES}/hello/Hello.edl")

# Define the target for the program build.
add_executable (${PROGRAM_NAME} "src/hello.c")
add_dependencies (${PROGRAM_NAME} ${PROGRAM_NAME}_edl_files)

# Pack the application into a KPA package
add_kpa_package (${PACKAGE} ID ${PROGRAM_NAME} MANIFEST_V "2")
add_kpa_component (${PACKAGE} RESOURCE ${PROGRAM_NAME} TYPE "bin")
add_kpa_component (${PACKAGE}
                   RUN_CONFIGURATION "default"
                   TYPE              "application"
                   NAME              "${PROGRAM_NAME}"
                   EIID              "hello.${PROGRAM_NAME}"
                   PATH              "bin/${PROGRAM_NAME}"
                   NO_AUTO_VFS_CONNECTION)
```

CMakeLists.txt file for building the solution image

The `CMakeLists.txt` file for building the solution image must contain the following commands:

- `include (${KL_SDK_ROOT_PATH}/share/vendor_rom/build_vendor_rom.cmake)` – connects the `CMake` library containing scripts for building the VendorROMFS image.
- Configure the packages of system programs and drivers that need to be included in the solution.
  - A package is connected by using the `find_package (<package name> [<parameter1>]...)` command.
  - For some packages, you must also set the values of properties by using the command `set_target_properties (<target>... PROPERTIES <property1> <value1> [<property2> <value2>]...)`.

> `CMake` descriptions of system programs and drivers provided in the KasperskyOS Community Edition SDK, and descriptions of their exported variables and properties are located in the corresponding files at `/opt/KasperskyOS-Community-Edition-<platform>-<version>/sysroot-*-kos/lib/cmake/<program name>/<program name>Config.cmake`

- build_vendor_romfs() command for building the VendorROMFS image.
- Commands for building the root file system rootfs.
- One or more commands for building the solution image:
  - build_sd_image () – creates the target for building an image for running a solution on the hardware platform.
  - generate_qemu_target () – creates the target for building a solution image for QEMU.

For examples of CMakeLists.txt files used to build a solution image, see the ./einit/CMakeLists.txt build scripts in the SDK examples.

product_init.yaml.in template

> You are advised to use KPA packages and the ServiceManager system program to run applications.

The KasperskyOS Community Edition SDK includes the kl.ProductRomfsLoader initializing program, which can be used to start programs that were not packed into KPA packages. The kl.ProductRomfsLoader program starts programs in accordance with the init description (the product_init.yaml file). This program startup method is not recommended but is used to start some system programs that do not support startup from KPA packages.

The `product_init.yaml.in` template is used to automatically generate a *part* of the `product_init.yaml` file with `CMake`.

When using the `product_init.yaml.in` template, you do not have to manually add descriptions of system programs and the IPC channels for connecting to them to the `product_init.yaml` file.

The `product_init.yaml.in` template must contain the following data:

- Root `entities` key.
- For programs that use an IPC mechanism, you must specify a list of IPC channels that connect this application to other applications.

  The `@INIT_<program name>_ENTITY_CONNECTIONS@` macro is used to specify a list of IPC channels that connect this program to the system programs included in KasperskyOS Community Edition SDK. During the build, this macro is replaced by a list of IPC channels with all system programs linked with the application. The `target` and `id` fields are filled according to the `connect.yaml` files from KasperskyOS Community Edition SDK located at `/opt/KasperskyOS-Community-Edition-<platform>-<version>/sysroot-*-kos/include/<system program name>`).

- The `@INIT_<program name>_ENTITY_CONNECTIONS@` macro also adds the list of connections for each program specified in the `EXTRA_CONNECTIONS` property when building this program.

- If you need to pass the arguments of the `main()` function specified in the `EXTRA_ARGS` property to a program when building this program, you must use the `@INIT_<program name>_ENTITY_ARGS@` macro. During the build, this macro is replaced by a list of arguments of the `main()` function as specified in the `EXTRA_ARGS` property.

- If you need to pass the values of environment variables specified in the `EXTRA_ENV` property to a program when building this program, you must use the `@INIT_<program name>_ENTITY_ENV@` macro. During the build, this macro is replaced by a dictionary of environment variables and their values as specified in the `EXTRA_ENV` property.

- `@INIT_EXTERNAL_ENTITIES@` – during the build, this macro is replaced with the list of system programs linked to the application and their IPC channels, `main()` function arguments, and values of environment variables.

Example product_init.yaml.in template

product_init.yaml.in

```
entities:

- name: kl.drivers.DNetSrv
  path: dnet_executable
  wait_for: ready

- name: kl.VfsNet
  path: VfsNet
  wait_for: ready
  after:
  - kl.drivers.DNetSrv
  env:
    RUMP_NETEN_NAME_POLICY: @VFSENV_RUMP_NETEN_NAME_POLICY@
    VFS_FILESYSTEM_BACKEND: "server:kl.VfsNet:relay:local:kl.VfsNet"
    VFS_NETWORK_BACKEND: "server:kl.VfsNet:relay:local:kl.VfsNet"
    VFS_RELAY_DESCENDANTS: "VfsFs=client:task:kl.VfsEntity"
    ALLOWED_FS_PATHS: "/etc/hosts;/etc/resolv.conf"
  args:
  - '-l'
  - 'devfs /dev devfs 0'
```

CMake libraries in KasperskyOS Community Edition SDK

This section contains a description of the libraries that are provided in KasperskyOS Community Edition SDK for automatically building a KasperskyOS-based solution.

platform library

The command is declared in the file named `/opt/KasperskyOS-Community-Edition-<platform>-<version>/toolchain/share/cmake/Modules/platform/common/platform.cmake`.

The `platform` library contains the following commands:

- `initialize_platform ([<parameters>])` – initializes the `platform` library.

  Must be used in the `CMakeLists.txt` root file.

  Specifies the way executable files are linked in a solution project:

  - `initialize_platform ()` – dynamic linking of executable files.
  - `initialize_platform (USE_DYNAMIC_LEGACY)` – dynamic linking of executable files. In contrast to the `initialize_platform ()` command without parameters, the `initialize_platform (USE_DYNAMIC_LEGACY)` command uses an outdated version of dynamic linking in which the executable file of the program is statically linked with the dynamic library loader (`libld.elf_so.a`) and the libraries `libkos`, `libc`, `libm`, `libpthread`, `libvfs`, `librtlcom`, and `libdynld`. When dynamic linking is enabled as a result of running the `initialize_platform ()` command without parameters, the dynamic library loader is statically linked with these libraries. This results is the creation of the `ld.elf_so` library, with which the program's executable file is dynamically linked.
  - `initialize_platform (FORCE_STATIC)` – forced static linking of executable files.

  You can specify the additional parameter `NO_NEW_VERSION_CHECK` (for example, `initialize_platform (FORCE_STATIC NO_NEW_VERSION_CHECK)`) to disable checking for SDK updates and transmitting the SDK version to the Kaspersky server. For more details about the data provision policy, see Data provision.

- `platform_target_force_static (<build target name>)` – forced static linking of the executable file specified via the `add_executable ()` command. Static linking is performed regardless of the linking method specified by the `initialize_platform ([<parameter>])` command.

- `platform_target_use_dynamic_legacy (<build target name>)` – dynamic linking of the executable file specified via the `add_executable ()` command. Dynamic linking is performed regardless of which linking method is defined by the `initialize_platform ([<parameter>]) command`. Enables an outdated version of dynamic linking such as the `initialize_platform (USE_DYNAMIC_LEGACY)` command, but only for one executable file.

- `project_header_default (<parameters>)` – sets the compiler and linker flags.

  Must be used in the `CMakeLists.txt` root file.

  Command parameters are defined as pairs consisting of a flag and its state (`YES` means it is set, `NO` means it is not set): `"FLAG_1:{YES|NO}"` `"FLAG_2:{YES|NO}"` … `"FLAG_N:{YES|NO}"`. The compiler flags that specify the C/C++ language standards and the compiler flag that controls the display of warnings are shown in the table below. If the compiler flag specifying the C/C++ language standards is not set, the ISO C17 and 2017 ISO C++ standards with GNU extensions are used by default.

  Example command:

```
project_header_default ("STANDARD_GNU_17:YES" "STRICT_WARNINGS:NO")
```

Information on some compiler flags

Flag	Meaning	Corresponding C compiler options	Corresponding C++ compiler options
`STANDARD_ANSI`	The ISO C90 and 1998 ISO C++ standards are used.	`-ansi`	`-ansi`
`STANDARD_C99`	The ISO C99 standard is used.	`-std=c99`	No
`STANDARD_GNU_C99`	The ISO C99 standard with GNU extensions is used.	`-std=gnu99`	No

Flag	Meaning	Corresponding C compiler options	Corresponding C++ compiler options
`STANDARD_11`	The ISO C11 and 2011 ISO C++ standards are used.	`-std=c11` or `-std=c1x`	`-std=c++11` or `-std=c++0x`
`STANDARD_GNU_11`	The ISO C11 and 2011 ISO C++ standards with GNU extensions are used.	`-std=gnu1x` or `-std=gnu11`	`-std=gnu++0x` or `-std=gnu++11`
`STANDARD_14`	The 2014 ISO C++ standard is used.	No	`-std=c++14`
`STANDARD_GNU_14`	The 2014 ISO C++ standard with GNU extensions is used.	No	`-std=gnu++14`
`STANDARD_17`	The ISO C17 and 2017 ISO C++ standards are used.	`-std=c17`	`-std=c++17`
`STANDARD_GNU_17`	The ISO C17 and 2017 ISO C++ standards with GNU extensions are used.	`-std=gnu17`	`-std=gnu++17`

`STRICT_WARNINGS`	Warning output are enabled for the detection of potential issues and errors in code written in C and C++.
	By default, these warnings are enabled.
		`-Wcast-qual`
		`-Wcast-align`
		`-Wundef`
		`-Wmissing-prototypes`
			`-Wcast-qual`
			`-Wcast-align`
			`-Wundef`

A full description of the flags can be found in the file `/opt/KasperskyOS-Community-Edition-<platform>-<version>/toolchain/share/cmake/Modules/platform/building/compiler_flags-GNU.cmake`.

nk library

This section contains information on CMake commands for creating EDL files and generating transport code in C and C++.

Creating EDL files and generating transport code in C

Here you can find information on CMake commands for creating EDL files of formal specifications of solution components and generating transport code in C.

generate_edl_file ()

```
generate_edl_file ("<TASK_CLASS_NO_PREF>"
                   [PREFIX "<PREF>"]
                   [EDL_ENDPOINTS "<ENDPOINT1>" ["<ENDPOINT2>"]...]
                   [EDL_COMPONENTS "<COMP_INST1>" ["<COMP_INST2>"]...]
                   [SECURITY "<SECURITY_IFACE>"]
                   [OUTPUT_DIR "<PATH>"])
```

Generates the EDL file of the formal specification of the solution component and saves the absolute path to the generated EDL file in the CMake variable `EDL_FILE`.

Parameters:

- `"<TASK_CLASS_NO_PREF>"` – name of the process class without a prefix.

  For example, if the process class name is defined as `test.PictureManager` or `test.ex.PictureManager`, you must specify `PictureManager`.

- `PREFIX "<PREF>"` – prefix in the process class name.

  For example, if the process class name is defined as `test.PictureManager`, `test` must be defined, and the `test.ex.PictureManager` process class name must be set to `test.ex`. The default value is `kl`. If you specify an empty string, the parameter will take the default value.

- `EDL_ENDPOINTS "<ENDPOINT1>" ["<ENDPOINT2>"]...` – endpoints.

  The `ENDPOINT<i>` values are constructs in the format `<endpoint name> : <interface name>`, for example: `write : secure_logger.Write`.

- `EDL_COMPONENTS "<COMP_INST1>" ["<COMP_INST2>"]...` – instances of components.

  The `COMP_INST<i>` values are constructs in the format `<component instance name> : <component name>`, for example: `controlimpl : ping.Control`.

- `SECURITY "<SECURITY_IFACE>"` – name of the security interface.

- `OUTPUT_DIR "<PATH>"` – path to the directory where the EDL file will be saved.

  You can specify an absolute path or a path relative to the directory corresponding to the CMake variable `CMAKE_CURRENT_BINARY_DIR`. By default, the parameter has the value of the CMake variable `CMAKE_CURRENT_BINARY_DIR`.

Example calls:

```
generate_edl_file ("Server"
                   PREFIX "ping"
                   EDL_COMPONENTS "controlimpl : ping.Control"
                   OUTPUT_DIR "../../resources/xdl/ping")
```

```
generate_edl_file ("OrchestratorLib"
                   PREFIX "orchestrator"
                   EDL_COMPONENTS "sm0 : kl.settings_manager.SM"
                   EDL_ENDPOINTS
                   "updateControl : orchestrator.UpdateControl"
                   "updateTaskControl : orchestrator.UpdateTaskControl"
                   OUTPUT_DIR "${CMAKE_BINARY_DIR}/_headers_")
```

For an example of using, see the article titled CMakeLists.txt files for building application software.

nk_build_idl_files ()

```
nk_build_idl_files (<IDL_H_TARGET_NAME>
                    IDL "<PATH_TO_IDL_FILE1>" ["<PATH_TO_IDL_FILE2>"]...
                    [NK_MODULE "<PREFIX>"]
                    [DEPENDS <DEPENDING_TARGET1> [<DEPENDING_TARGET2>]...]
                    [NK_FLAGS "<PARAM1>" ["<PARAM2>"]...])
```

Creates a CMake target for generating `*.idl.h` files from the formal specification of a solution component.

Parameters:

- `<IDL_H_TARGET_NAME>` – name of the CMake target for generating `.idl.h` files.

- `IDL "<PATH_TO_IDL_FILE1>" ["<PATH_TO_IDL_FILE2>"]...` – paths to IDL files.

The formal specification of a solution component may include several IDL files, including those supplied as part of the KasperskyOS Community Edition SDK. You need to execute either one call of the CMake command `nk_build_idl_files ()` specifying the entire set of IDL files (except the ones supplied, because `*.idl.h` files generated from them are also provided), or execute multiple calls specifying subsets from this set. Multiple calls are required if the interface names have different prefixes. For example, the interface names `test.ex.Get`, `test.hi.Set`, and `test.Clear` have different prefixes and correspond to different values of the `NK_MODULE` parameter, therefore you cannot specify IDL files for these interfaces in a single call of the CMake command `nk_build_idl_files ()`.

The order in which the IDL files are specified in the parameter matters if there are dependencies between these IDL files. (An IDL file depends on another IDL file if the `import` declaration in the first IDL file contains the name of the package described in the second IDL file.) You must ensure that the IDL file on which the other IDL file depends is specified before the dependent IDL file.

You can specify absolute paths or paths relative to the directory containing the `CMakeLists.txt` file that calls the CMake command `nk_build_idl_files ()`.

- `NK_MODULE "<PREFIX>"` – value obtained after excluding the IDL file name from the package name and replacing the `.` character with the `/` character.

  For example, if the package name is specified as `Get` in the IDL file, the test `""` must be defined; the `test` value must be specified for the `test.Get` package name; and the `test/ex` value must be specified for the `test.ex.Set` package name. The default value is `kl`.

- `DEPENDS <DEPENDING_TARGET1> [<DEPENDING_TARGET2>]...` – names of CMake targets that must be executed before generating `*.idl.h` files.

  In this parameter, you need to specify the names of the CMake targets that are created by the CMake command `nk_build_idl_files ()` for those IDL files on which the IDL files specified in the `IDL` parameter depend. (An IDL file depends on another IDL file if the `import` declaration in the first IDL file contains the name of the package described in the second IDL file.) This applies only to IDL files that are not provided in the KasperskyOS Community Edition SDK. For IDL files that are provided in the KasperskyOS Community Edition SDK, you do not need to call the CMake command `nk_build_idl_files ()` because the `*.idl.h` files generated from these IDL files are also provided.

- `NK_FLAGS "<PARAM1>" ["<PARAM2>"]...` are transport code generation parameters.

Example calls:

```
nk_build_idl_files (echo_idl_files
                    NK_MODULE "echo"
                    IDL "../resources/xdl/echo/Ping.idl"
                    NK_FLAGS "--extended-errors" "--trace-client-ipc=headers")
```

```
nk_build_idl_files (idl_files
                    NK_MODULE "kl/authenticator"
                    IDL "${CMAKE_SOURCE_DIR}/resources/Retcode.idl"
                    "${CMAKE_SOURCE_DIR}/resources/Types.idl"
                    "${CMAKE_SOURCE_DIR}/resources/SessionManager.idl"
                    "${CMAKE_SOURCE_DIR}/resources/Authentication.idl"
                    "${CMAKE_SOURCE_DIR}/resources/ComponentManager.idl"
                    "${CMAKE_SOURCE_DIR}/resources/UserManager.idl"
                    NK_FLAGS "--extended-errors")
```

```
nk_build_idl_files (idl_files_1
                    IDL "Block.idl"
                    NK_MODULE "kl/drivers")

nk_build_idl_files (idl_files_2
                    IDL "Security.idl"
```

```
                    NK_MODULE "kl/drivers/block"
                    DEPENDS idl_files_1)
```

For an example of using, see the article titled CMakeLists.txt files for building application software.

nk_build_cdl_files ()

```
nk_build_cdl_files (<CDL_H_TARGET_NAME>
                    CDL "<PATH_TO_CDL_FILE1>" ["<PATH_TO_CDL_FILE2>"]...
                    [NK_MODULE "<PREFIX>"]
                    [DEPENDS <DEPENDING_TARGET1> [<DEPENDING_TARGET2>]...]
                    [NK_FLAGS "<PARAM1>" ["<PARAM2>"]...])
```

Creates a CMake target for generating `*.cdl.h` files from the formal specification of a solution component.

Parameters:

- `<CDL_H_TARGET_NAME>` – name of the CMake target for generating `*.cdl.h` files.
- `CDL "<PATH_TO_CDL_FILE1>" ["<PATH_TO_CDL_FILE2>"]...` – paths to CDL files.

  The formal specification of a solution component may include several CDL files, including those supplied as part of the KasperskyOS Community Edition SDK. You need to execute either one call of the CMake command `nk_build_cdl_files ()` specifying the entire set of CDL files (except the ones supplied, because the `*.cdl.h` files generated from them are also provided), or execute multiple calls specifying subsets from this set. Multiple calls are required in the following cases:

  - The component names have different prefixes.

    For example, the component names `test.ex.Control`, `test.hi.View`, and `test.Calc` have different prefixes and correspond to different values of the `NK_MODULE` parameter, therefore you cannot specify CDL files for these components in a single call of the CMake command `nk_build_cdl_files ()`.

  - There are dependencies between CDL files.

    A CDL file depends on another CDL file if the `components` declaration in the first CDL file includes an instance of the component described in the second CDL file. In this case, you must ensure that the CMake command `nk_build_cdl_files ()` specifying the second CDL file is executed before executing this CMake command specifying the first CDL file. To ensure the necessary order of calls of the CMake command `nk_build_cdl_files ()`, you must use the `DEPENDS` parameter.

  You can specify absolute paths or paths relative to the directory containing the `CMakeLists.txt` file that calls the CMake command `nk_build_cdl_files ()`.

- `NK_MODULE "<PREFIX>"` – value obtained after excluding the CDL file name from the component name and replacing the `.` character with the `/` character.

  For example, if the component name is specified as `Control` in the CDL file, the `""` value must be defined; the `test` value must be specified for the `test.Control` component name; and the `test/ex` value must be specified for the `test.ex.Control` component name. The default value is `kl`.

- `DEPENDS <DEPENDING_TARGET1> [<DEPENDING_TARGET2>]...` – names of CMake targets that must be executed before generating `*.cdl.h` files.

  In this parameter, you need to specify the names of the CMake targets that are created by the CMake command `nk_build_cdl_files ()` and/or the CMake command `nk_build_idl_files ()` for those CDL files and/or IDL files on which the CDL files specified in the `CDL` parameter depend. (A CDL file depends on another CDL file if the `components` declaration in the first CDL file includes an instance of the component described in the second CDL file. The CDL file depends on an IDL file if the `endpoints` declaration in the CDL file includes an endpoint with the interface described in the IDL file.) This applies only to CDL and IDL files that are not provided in the KasperskyOS Community Edition SDK. For CDL and IDL files that are provided in the KasperskyOS Community Edition SDK, you do not need to call the CMake commands `nk_build_cdl_files ()` and `nk_build_idl_files ()` because the `*.cdl.h` and `*.idl.h` files generated from these CDL and IDL files are also provided.

- `NK_FLAGS "<PARAM1>" ["<PARAM2>"]...` are transport code generation parameters.

Example call:

```
nk_build_cdl_files (echo_cdl_files
                    DEPENDS echo_idl_files
                    NK_MODULE "echo"
                    CDL "../resources/xdl/echo/Ping.cdl")
```

For an example of using, see the article titled CMakeLists.txt files for building application software.

nk_build_edl_files ()

```
nk_build_edl_files (<EDL_H_TARGET_NAME>
                    EDL "<PATH_TO_EDL_FILE>"
                    [NK_MODULE "<PREFIX>"]
                    [DEPENDS <DEPENDING_TARGET1> [<DEPENDING_TARGET2>]...]
                    [NK_FLAGS "<PARAM1>" ["<PARAM2>"]...])
```

Creates a CMake target for generating an `*.edl.h` file from the formal specification of a solution component.

Parameters:

- `<EDL_H_TARGET_NAME>` – name of the CMake target for generating an `*.edl.h` file.

- `EDL "<PATH_TO_EDL_FILE>"` – path to the EDL file.

  You can specify an absolute path or a path relative to the directory containing the `CMakeLists.txt` file that calls the CMake command `nk_build_edl_files ()`.

- `NK_MODULE "<PREFIX>"` – value obtained after excluding the EDL file name from the process class name and replacing the `.` character with the `/` character.

  For example, if the process class name is set to `PictureManager` in the EDL file, the `""` value must be defined; the `test.PictureManager` process class name must be set to `test`; and the `test.ex.PictureManager` process class name must be set to `test/ex`. The default value is `kl`.

- `DEPENDS <DEPENDING_TARGET1> [<DEPENDING_TARGET2>]...` – names of CMake targets that must be executed before generating an `*.edl.h` file.

  In this parameter, you need to specify the names of the CMake targets that are created by the CMake command `nk_build_cdl_files ()` and/or the CMake command `nk_build_idl_files ()` for those CDL files and/or IDL files on which the EDL file depends. (An EDL file depends on a CDL file if the `components` declaration in the EDL file includes an instance of the component described in the CDL file. The EDL file depends on an IDL file if the `endpoints` declaration in the EDL file includes an endpoint with the interface described in the IDL file.) This applies only to CDL and IDL files that are not provided in the KasperskyOS Community Edition SDK. For CDL and IDL files that are provided in the KasperskyOS Community Edition SDK, you do not need to call the CMake commands `nk_build_cdl_files ()` and `nk_build_idl_files ()` because the `*.cdl.h` and `*.idl.h` files generated from these CDL and IDL files are also provided.

- `NK_FLAGS "<PARAM1>" ["<PARAM2>"]...` are transport code generation parameters.

Example calls:

```
nk_build_edl_files (echo_server_edl_files
                    DEPENDS echo_cdl_files
                    NK_MODULE "echo"
                    EDL "../resources/xdl/echo/Server.edl"
                    NK_FLAGS "--extended-errors")
```

```
nk_build_edl_files (echo_client_edl_files
                    NK_MODULE "echo"
                    EDL "../resources/xdl/echo/Client.edl")
```

For an example of using, see the article titled CMakeLists.txt files for building application software.

Parameters for generating transport code in C

You can use the `NK_FLAGS` parameter of the CMake commands `nk_build_*dl_files ()` to set the parameters for generating transport code in C.

Configuration necessary for using interface methods with error parameters

If the IDL file includes at least one signature of an interface method containing at least one error parameter, you must specify one of the following parameters:

- `--extended-errors`

  This parameter provides the capability to use interface methods with one or more error parameters of user-defined IDL types. The client works with error parameters like it works with output parameters.

- `--deprecated-no-extended-errors`

  This parameter provides the capability to use interface methods only with one `status` error parameter of the IDL type `UInt16` whose value is received by the client via return code of the interface method.

  > The method for passing error parameters corresponding to the `--deprecated-no-extended-errors` parameter is obsolete and will no longer be supported in the future, therefore it is recommended to specify the `--extended-errors` parameter.

Selective generation of transport code

To reduce the volume of generated transport code, you can use flags for selective generation of transport code. For example, you can use the `--server` flag for programs that provides endpoints, and use the `--client` flag for programs that utilize the endpoints.

Flags for selective generation of transport code for an IDL file:

- `--types`

  The transport code includes the types corresponding to the IDL types from the defined IDL file, and the types corresponding to this file's imported IDL types that are used in IDL types of the defined IDL file. However, the types corresponding to imported IDL constants and to the aliases of imported IDL types are not included in the `*.idl.h` file. To use the types corresponding to imported IDL constants and to the aliases of imported IDL types, you must separately generate the transport code for the IDL files from which you are importing.

- `--interface`

  The transport code corresponds to the `--types` flag, and includes the types of structures of the constant part of IPC requests and IPC responses for interface methods whose signatures are specified in the defined IDL file. In addition, the transport code contains constants indicating the sizes of IPC message arenas.

- `--client`

  The transport code corresponds to the `--interface` flag, and includes the proxy object type, proxy object initialization method, and the interface methods specified in the defined IDL file.

- `--server` (default)

  The transport code corresponds to the `--interface` flag, and includes the types and dispatcher (dispatch method) used to process IPC requests corresponding to the interface methods specified in the defined IDL file.

Flags for selective generation of transport code for a CDL or EDL file:

- `--types`

  The transport code includes the types corresponding to the IDL types that are used in the method parameters of endpoints provided by the component (for the defined CDL file) or process class (for the defined EDL file).

- `--endpoints`

  The transport code corresponds to the `--types` flag, and includes the types of structures of the constant part of IPC requests and IPC responses for the methods of endpoints provided by the component (for the defined CDL file) or process class (for the defined EDL file). In addition, the transport code contains constants indicating the sizes of IPC message arenas.

- `--client`

  The transport code corresponds to the `--endpoints` flag and includes the proxy object types, proxy object initialization methods, and methods of endpoints provided by the component (for the defined CDL file) or process class (for the defined EDL file).

- `--server` (default)

  The transport code corresponds to the `--endpoints` flag, and includes the types and dispatchers (dispatch methods) used to process IPC requests corresponding to the endpoints provided by the component (for the defined CDL file) or process class (for the defined EDL file). In addition, the transport code includes the stub types and stub initialization methods.

Printing diagnostic information about sending and receiving IPC messages

Transport code can generate diagnostic information about sending and receiving IPC messages and print this data via standard error. To generate transport code with these capabilities, use the following parameters:

- `--trace-client-ipc {headers|dump}`

  The code for printing diagnostic information is executed directly before the `Call()` system call is executed and immediately after it is executed. If the `headers` value is specified, the diagnostic information includes the endpoint method ID (MID), the Runtime Implementation Identifier (RIID), the size of the constant part of the IPC message (in bytes), the contents of the IPC message arena handle, the size of the IPC message arena (in bytes), and the size of the utilized part of the IPC message arena (in bytes). If the `dump` value is specified, the diagnostic information additionally includes the contents of the constant part and arena of the IPC message in hexadecimal format.

  When using this parameter, you must either specify the selective generation flag for transport code `--client` or refrain from specifying a selective generation flag for transport code.

- `--trace-server-ipc {headers|dump}`

  The code for printing diagnostic information is executed directly before calling the function that implements the interface method, and immediately after the completion of this function. In other words, it is executed when the dispatcher (dispatch method) is called in the interval between execution of the `Recv()` and `Reply()` system calls. If the `headers` value is specified, the diagnostic information includes the endpoint method ID (MID), the Runtime Implementation Identifier (RIID), the size of the constant part of the IPC message (in bytes), the contents of the IPC message arena handle, the size of the IPC message arena (in bytes), and the size of the utilized part of the IPC message arena (in bytes). If the `dump` value is specified, the diagnostic information additionally includes the contents of the constant part and arena of the IPC message in hexadecimal format.

  When using this parameter, you must either specify the selective generation flag for transport code `--server` or refrain from specifying a selective generation flag for transport code.

- `--ipc-trace-method-filter <METHOD 1>[,<METHOD 2>]...`

  Diagnostic information is printed if only the defined interface methods are called. For the `METHOD` value, you can use the interface method name or the construct `<package name>:<interface method name>`. The package name and interface method name are specified in the IDL file.

  If this parameter is not specified, diagnostic information is printed when any interface method is called.

  This parameter can be specified multiple times. For example, you can specify all required interface methods in one parameter or specify each required interface method in a separate parameter.

Managing warnings and errors

You can disable output of certain warnings of the transport code generator, or vice versa, assign these warnings as errors. To disable warning output, use the parameter `-Wno-<WARN_CODE>`. To assign a warning as an error, use the parameter `-Werror=<WARN_CODE>`. `WARN_CODE` can take the following values:

- `xdl.component-named-as-task-class`

  The process class name in an EDL file matches the component name in the `component` declaration of a CDL file that is not linked to this EDL file. In other words, in the defined directories, a search of the transport code generator detected a CDL file in which the `component` declaration contains a component name matching the process class name in the EDL file. Moreover, the `components` declaration of the EDL file does not contain a component name matching the found CDL file, or the EDL file does not contain a `components` declaration at all.

- `xdl.deprecated-syntax`

  An obsolete `entity` declaration is used in the EDL file.

- `xdl.multiple-files`

  Transport code generator detected multiple IDL files matching the name of an interface or package, or multiple CDL files matching a component name. For example, in the defined directories, a search of the transport code generator detected multiple CDL files that match one component name specified in the `components` declaration of an EDL file, or multiple IDL files that match one interface name specified in the `endpoints` declaration of a CDL file. (To generate transport code, the first suitable IDL or CDL file among the several that are found is used.)

Generating transport code in C++

Here you can find information on CMake commands for generating transport code in C++.

add_nk_idl ()

```
add_nk_idl (NAME IDL_FILE LANG ...)
```

Creates a `CMake` target for generating a `*.idl.cpp.h` header file for a defined IDL file. The command also creates a library containing the transport code for the defined interface. To link to this library, use the bind_nk_targets () command.

The generated header files contain a C++ representation for the interface and data types described in the IDL file, and the methods required for use of proxy objects and stubs.

Parameters:

- `NAME` – name of the `CMake` target. Required parameter.
- `IDL_FILE` – path to the IDL file. Required parameter.
- `NK_MODULE` – value obtained after excluding the IDL file name from the package name and replacing the `.` character with the `/` character.

  For example, if the package name is specified as `Get` in the IDL file, the test `""` must be defined; the `test` value must be specified for the `test.Get` package name; and the `test/ex` value must be specified for the `test.ex.Set` package name. The default value is `kl`.
- `NOINSTALL` – if this option is specified, files will only be generated in the working directory and will not be installed in global directories: `${CMAKE_BINARY_DIR}/_headers_` `${CMAKE_BINARY_DIR}/_headers_/${PROJECT_NAME}`.
- `WORKING_DIRECTORY` – working directory for calling the transport code generator, which is `${CMAKE_CURRENT_BINARY_DIR}` by default.
- `LANG` – parameter in which you need to specify the `CXX` value. Required parameter.

Example call:

```
add_nk_idl (ANIMAL_IDL "${CMAKE_SOURCE_DIR}/resources/Animal.idl"
            NK_MODULE  "example"
            LANG       "CXX")
```

add_nk_cdl ()

```
add_nk_cdl (NAME CDL_FILE LANG ...)
```

Creates a `CMake` target to generate a `*.cdl.cpp.h` file for the specified CDL file. The command also creates a library containing the transport code for the defined component. To link to this library, use the bind_nk_targets () command.

The `*.cdl.cpp.h` file contains the tree of embedded components and endpoints provided by the component described in the CDL file.

Parameters:

- `NAME` – name of the `CMake` target. Required parameter.
- `CDL_FILE` – path to the CDL file. Required parameter.
- `NK_MODULE` – value obtained after excluding the CDL file name from the component name and replacing the `.` character with the `/` character.

  For example, if the component name is specified as `Control` in the CDL file, the `""` value must be defined; the `test` value must be specified for the `test.Control` component name; and the `test/ex` value must be specified for the `test.ex.Control` component name. The default value is `kl`.
- `NOINSTALL` – if this option is specified, files will only be generated in the working directory and will not be installed in global directories: `${CMAKE_BINARY_DIR}/_headers_ ${CMAKE_BINARY_DIR}/_headers_/${PROJECT_NAME}`.
- `WORKING_DIRECTORY` – working directory for calling the transport code generator, which is `${CMAKE_CURRENT_BINARY_DIR}` by default.
- `LANG` – parameter in which you need to specify the `CXX` value. Required parameter.

Example call:

```
add_nk_cdl (CAT_CDL    "${CMAKE_SOURCE_DIR}/resources/Cat.cdl"
            NK_MODULE "example"
            LANG      "CXX")
```

add_nk_edl ()

```
add_nk_edl (NAME EDL_FILE LANG ...)
```

Creates a `CMake` target to generate an `*.edl.cpp.h` file for the specified EDL file. The command also creates a library containing the transport code for the server or client program. To link to this library, use the bind_nk_targets () command.

The `*.edl.cpp.h` file contains the tree of embedded components and endpoints provided by the process class described in the EDL file.

Parameters:

- `NAME` – name of the `CMake` target. Required parameter.
- `EDL_FILE` – path to the EDL file. Required parameter.
- `NK_MODULE` – value obtained after excluding the EDL file name from the process class name and replacing the `.` character with the `/` character.

  For example, if the process class name is set to `PictureManager` in the EDL file, the `""` value must be defined; the `test.PictureManager` process class name must be set to `test`; and the `test.ex.PictureManager` process class name must be set to `test/ex`. The default value is `kl`.
- `NOINSTALL` – if this option is specified, files will only be generated in the working directory and will not be installed in global directories: `${CMAKE_BINARY_DIR}/_headers_ ${CMAKE_BINARY_DIR}/_headers_/${PROJECT_NAME}`.
- `WORKING_DIRECTORY` – working directory for calling the transport code generator, which is `${CMAKE_CURRENT_BINARY_DIR}` by default.
- `LANG` – parameter in which you need to specify the `CXX` value. Required parameter.

Example call:

```
add_nk_edl (SERVER_EDL "${CMAKE_SOURCE_DIR}/resources/Server.edl"
            NK_MODULE  "example"
            LANG       "CXX")
```

bind_nk_targets ()

```
bind_nk_targets (<dependent target> <dependency1> <dependency2> ...)
```

Links the `CMake` targets created by the `CMake` commands add_nk_idl (), add_nk_cdl (), and add_nk_edl () to the main `CMake` target of the program build. The command accepts at least two parameters. The first parameter is the main `CMake` target of the build, and the remaining parameters are the `CMake` targets on which it depends. The binding process results in the following:

1. The order in which the `CMake` targets are built is determined. If there are several dependencies, one command call can be used only if the specified dependencies are independent of each other. If one of the dependencies uses packages from another dependency, they must first be bound to each other as in the example below. Ignoring the order of dependencies when binding targets can lead to race conditions when building an application.
2. The main build target will be automatically linked to the libraries resulting from the build of dependent targets.

Example call:

```
add_nk_idl (WRITER_SERVER_IDL
    "${RESOURCE_DIR}/Writer.idl"
    NK_MODULE "example"
    LANG "CXX")

add_nk_edl (SERVER_EDL
    "${RESOURCE_DIR}/Client.edl"
    NK_MODULE "example"
    LANG "CXX")
```

```
bind_nk_targets(SERVER_EDL WRITER_SERVER_IDL)
bind_nk_targets(Server SERVER_EDL)
```

kpa library

This section contains a description of commands of the `CMake` library `kpa`, which is intended for building KPA packages.

add_kpa_package()

This command is declared in the file named `/opt/KasperskyOS-Community-Edition-<platform>-<version>/toolchain/share/cmake/Modules/platform/kpa.cmake`.

```
add_kpa_package(KPA_TARGET_NAME ...)
```

The command creates a `CMake` target for building the KPA package. When building this target, the build process automatically creates the KPA package manifest and the KPA package, which includes all components that were added to this target using the `CMake` command add_kpa_component().

Parameters:

- `KPA_TARGET_NAME` – name of the `CMake` target. Required parameter.
- `VERSION` – version of the KPA package. The default value is `PROJECT_VERSION`.
- `ID` – ID of the KPA package. The default value is `KPA_TARGET_NAME`.
- `DEVELOPER_ID` or `DEVELOPER` – developer ID. The default value is `"unspecified"`.
- `BUILD_TYPE` – package variant depending on the build type. The build type is defined by the `CMAKE_BUILD_TYPE` variable.
- `DONT_VERIFY` – if this parameter is defined, verification of the presence of all KPA package components specified in its manifest and the absence of unspecified components, and calculation of the checksums of KPA package components and their comparison with those specified in the KPA package manifest will not be performed.
- `COMPRESS` – if specified, the KPA package will be built in compressed format.
- `MANIFEST_V` – version of the KPA package manifest. The default value: `"1"`.
- `MANIFEST` – path to the design-time manifest that has an empty `components` section. Only available for manifest version 2 or later.
- `PROVIDED_RUNNERS` – list of runners that come with the package. Only available for manifest version 2 or later.

Properties of the created `CMake` target:

- `KPA_OUTPUT_DIR` – directory in which the KPA package will be put.
- `KPA_OUTPUT_NAME` – name of the KPA file without the file extension. The default value is `${DEVELOPER_ID}.${ID}`.

Example call:

```
include (platform/kpa)

add_kpa_package (Hello.kpa ALL ID "hello" DEVELOPER_ID test_dev MANIFEST_V "2")

set_target_properties(Hello.kpa PROPERTIES
                      KPA_OUTPUT_NAME "hello")
```

add_kpa_component()

This command is declared in the file named `/opt/KasperskyOS-Community-Edition-<platform>-<version>/toolchain/share/cmake/Modules/platform/kpa.cmake`.

```
add_kpa_component(PACKAGE_NAME MODE ...)
```

The command adds the component to the specified `CMake` target for building the KPA package. This command can be called multiple times for the same KPA package build target. A KPA package build target is created by the add_kpa_package() command.

Parameters:

- `PACKAGE_NAME` – name of the `CMake` target for building the KPA package. Required parameter.
- `MODE` – type of component. Required parameter. The following values are possible:
  - `LIBRARY` – the component is a dynamic library. You must specify the full path to the file or the name of the `CMake` target. It is put into the directory `/<package_name>/lib` when the KPA package is installed.
  - `FILES` – the component is a file. You must specify the full path to the file. When a KPA package is installed, it is put into the `/<package_name>/res` directory.
  - `DIRECTORY` – the component is a directory containing files. You must specify the full path to the directory. When a KPA package is installed, it is put into the `/<package_name>/res` directory.
  - `RUN_CONFIGURATION` – the component is a program run configuration. You must specify the following parameters:
    - `ID` – startup configuration ID that is unique within the KPA package. Required parameter.
    - `NAME` – startup configuration name. Required parameter.
    - `TYPE` – type of run configuration: `"gui"` – process with a graphical user interface or `"service"` – endpoint process. Required parameter.

- `PATH` – full path to the executable file of the component after KPA package installation relative to the `/<package_name>` directory.
- `EIID` – security class of the program. Required parameter.
- `PRIMARY` – indicates whether the startup configuration is the primary one during program startup.
- `ARGS` – list of command-line arguments in the form of a string array for starting the program.
- `ENV` – list of environment variables for starting the program.

  If the program being started uses file systems (via the VFS component), you must specify the value `VFS_FILESYSTEM_BACKEND=client:kl.VfsSdCardFs` in the `ENV` field.

Example call with the RUN_CONFIGURATION parameter

```
include (platform/kpa)
Adding program startup configuration
add_kpa_component(Hello.kpa RUN_CONFIGURATION "app" NAME "Hello app" TYPE "service" EIID "hello.Hello" PATH
"bin/Hello")
```

- `RESOURCE` – used for adding arbitrary data to the KPA package. You must specify the following parameters:
  - Full path to the file or directory containing the data. Required parameter.
  - In the `TYPE` parameter, specify the data type: `"res"`, `"bin"`, `"lib"` or `"manifestLocale"`. Required parameter. For more details, refer to List of "components" objects.
  - (Optional) In the `DIRECTORY_PATH` parameter, specify the path to the directory where the data will be put after KPA package installation; the specified path should be relative to the `/<package_name>/res` directory.

Example call with the RESOURCE parameter

```
include (platform/kpa)
# Add the program executable file
add_kpa_component(Hello.kpa RESOURCE ".build/*-kos-debug/hello/Hello" TYPE "bin" DIRECTORY_PATH "bin/")
```

- `EXTENSION` – used to add a dependency between packages. You must specify the following parameters:
  - Name of the dependency.
  - In the `VALUE` parameter, specify the dependency package manifest string.

Example call with the EXTENSION parameter

```
include (platform/kpa)
# Adds a dependency
add_kpa_component (linked_loader_kpa EXTENSION "usedPackages"
    VALUE "{\"lib_provider.LibProvider\":{\"version\":\"1.*.*\",\"allowDynamicLibraries\":true}}")
```

build_sd_image library

The `build_sd_image` library is included in the KasperskyOS SDK and contains the `build_sd_image()` command. This command is declared in the file named `/opt/KasperskyOS-Community-Edition-<platform>-<version>/common/build-sd-image.cmake`.

```
build_sd_image(IMAGE_NAME ...)
```

The command creates a CMake target for building an SD card image for running a solution on the hardware platform.

Parameters:

- `IMAGE_NAME` – name of the CMake target for building an SD card image for running the solution on the hardware platform. Required parameter.
- `KOS_IMAGE_TARGET` – name of the CMake target for building a solution image for the hardware platform that will be added to the SD card image.
- `KOS_VENDOR_IMAGE` – path to the VendorROMFS image that will be included in the SD card image.
- `PATH_PARTITION1_IMAGE` – path to the first partition of the file system that will be used for the SD card image.
- `OUTPUT_IMAGE_NAME` – name of the SD card image.
- `DISK_SIZE` – size of the created SD card image in megabytes. Default value: 1024 MB.
- `BOOTARGS` – list of arguments that will be passed to the bootloader in the `--bootargs` parameter. It can be used to specify kernel startup parameters. For more details, refer to Using the command line of the KasperskyOS kernel.
- `PARTITION_CMD` – set of parameters that will be used to create and configure partitions in the SD card image. Separate the parameters with a space.
- `PREPACK_CMD` – command that will be executed before packing the file system into an image.
- `FDT_OVERLAYS` – list of DTBO overlays to be applied when building the solution.

Example call:

```
include (${KL_SDK_ROOT_PATH}/common/build-sd-image.cmake)

build_sd_image (sd-image
                KOS_IMAGE_TARGET  kos-image
                PATH_PARTITION1_IMAGE boot_dir
                OUTPUT_IMAGE_NAME ${DISK_IMAGE}
                PARTITION_CMD     -p1 ${IMAGE_PART1_FS} -%1 50 -f1 fat32
                                  -p2 ${IMAGE_PART2_FS} -%2 50 -f2 ext4)
```

minimal_kos library

The `minimal_kos` library is included in the KasperskyOS SDK and contains the commands and variables required for building the root file system rootfs and a solution to run in the QEMU emulator. Commands and library variables are declared in the file named `/opt/KasperskyOS-Community-Edition-<platform>-<version>/toolchain/share/cmake/Modules/platform/minimal_kos.cmake`.

generate_qemu_target()

This command is declared in the file named `/opt/KasperskyOS-Community-Edition-<platform>-<version>/toolchain/share/cmake/Modules/platform/minimal_kos.cmake`.

```
generate_qemu_target (TARGET_NAME ...)
```

This command creates a CMake target for building a solution image to run in QEMU.

Parameters:

- `TARGET_NAME` – name of the `CMake` target for building a solution image. Required parameter.
- `DISABLE_GRAPHICS` – flag to disable graphics when QEMU starts; adds `-nographic -monitor none` flags when QEMU starts.
- `KERNEL_IMAGE` – path to the KasperskyOS boot image. You must specify the `${MINIMAL_KOS_IMAGE}` variable.
- `VENDOR_IMAGE` – name of the CMake target for building a VendorROMFS image. It is defined when calling the build_vendor_romfs () command. Required parameter.
- `ROOTFS_IMAGE` – name of the CMake target for building the root file system rootfs. It is defined when calling the build_rootfs () command. Required parameter.
- `QEMU_FLAGS` – additional flags for running QEMU.
- `QEMU_ENV` – additional environment variables for starting QEMU.
- `QEMU_USER_NETWORK_FLAGS` – flags for setting network parameters when starting QEMU.
- `KERNEL_CMDLINE` – list of kernel startup parameters. For more details, refer to Using the command line of the KasperskyOS kernel.
- `DEPENDS <DEPENDING_TARGET1> [<DEPENDING_TARGET2>]...` – names of CMake targets that must be executed before starting this target.

Created targets:

- `${TARGET_NAME}` – main target for building a solution image.
- `sim` – target for running the image in QEMU.

Example call:

```
generate_qemu_target (qemu
                      KERNEL_IMAGE ${MINIMAL_KOS_IMAGE}
                      VENDOR_IMAGE vendor_romfs
                      ROOTFS_IMAGE rootfs)
```

build_rootfs()

This command is declared in the file named `/opt/KasperskyOS-Community-Edition-<platform>-<version>/toolchain/share/cmake/Modules/platform/minimal_kos.cmake`.

```
build_rootfs (TARGET_NAME ...)
```

This command creates a CMake target for creating an image of the root file system rootfs.

Parameters:

- `TARGET_NAME` – name of the `CMake` target for the build. Required parameter.
- `DIR` – directory where the image of the root file system will be built. Required parameter.
- `IMAGE_NAME` – name of the created image file.

Example call:

```
build_rootfs (rootfs DIR ${ROOTFS_DIR} DEPENDS rootfs_part_src)
```

build_einit()

This command is declared in the file named `/opt/KasperskyOS-Community-Edition-<platform>-<version>/toolchain/share/cmake/Modules/platform/minimal_kos.cmake`.

```
build_einit (TARGET_NAME)
```

This command creates a CMake target for creating the initializing program that is used to start programs that for some reason cannot be packed into KPA packages. You are advised to use this approach only if it is absolutely necessary. For more details, see the examples included in the SDK.

Parameters:

- `TARGET_NAME` – name of the `CMake` target for the build. Required parameter.
- `INIT_YAML` – path to the product_init.yaml file used to generate the program. Required parameter.
- `DEPENDS <DEPENDING_TARGET1> [<DEPENDING_TARGET2>]...` – names of CMake targets that must be executed before starting this target.

Example call:

```
set (PRODUCT_ROM_PROGRAMS dnet_executables::whole precompiled_vfs::VfsNet)
set (PRODUCT_INIT_YAML "${CMAKE_CURRENT_BINARY_DIR}/product_init.yaml")
build_einit (product_einit ${PRODUCT_INIT_YAML} DEPENDS ${PRODUCT_ROM_PROGRAMS})
```

build_romfs()

This command is declared in the file named `/opt/KasperskyOS-Community-Edition-<platform>-<version>/toolchain/share/cmake/Modules/platform/minimal_kos.cmake`.

```
build_romfs (TARGET_NAME ...)
```

This command creates a CMake target to create an image containing the initializing program for starting programs that for some reason cannot be packed into KPA packages. You are advised to use this approach only if it is absolutely necessary. For more details, see the examples included in the SDK.

Parameters:

- `TARGET_NAME` – name of the `CMake` target for the build. Required parameter.
- `EINIT` – name of the CMake target for generating the initializing program. It is defined when calling the build_einit () command. Required parameter.
- `FILES` – names of CMake targets for building programs that for some reason cannot be packed into KPA packages and will be added to the image. Required parameter.
- `DEPENDS <DEPENDING_TARGET1> [<DEPENDING_TARGET2>]...` – names of CMake targets that must be executed before starting this target.

Example call:

```
set (PRODUCT_ROM_PROGRAMS dnet_executables::whole precompiled_vfs::VfsNet)
build_romfs (product_romfs product_einit FILES ${PRODUCT_ROM_PROGRAMS})
```

build_vendor_rom library

The `build_vendor_rom` library is included in the KasperskyOS SDK and contains the `build_vendor_romfs()` command. This command is declared in the file named `/opt/KasperskyOS-Community-Edition-<platform>-<version>/share/vendor_rom/build_vendor_rom.cmake`.

```
build_vendor_romfs(VENDOR_ROM_TARGET ...)
```

The command creates a CMake target for building an additional image (VendorROMFS), which contains additional system programs and drivers required to work with the emulator or a specific hardware platform on which the solution will run. For more details, refer to Architecture and startup of a KasperskyOS-based solution.

Parameters:

- `VENDOR_ROM_TARGET` – name of the CMake target for building a VendorROMFS image. Required parameter.
- `WITH_PRODUCT_ROMFS` – if this parameter is specified, the solution will use the kl.ProductRomfsLoader program, which is used to start programs that for some reason cannot be packed into KPA packages. You are advised to use this approach only if it is absolutely necessary. For more details, see the examples included in the SDK.
- `SECURITY_PSL_FILE` – path to the security.psl file. Required parameter.
- `EDL_TARGETS` – names of the `CMake` targets on which the security module build depends. You must pass build targets of EDL files for all applications. The targets passed in this parameter will be built before starting the build of the security module. Required parameter.
- `GINGER_DEFINITIONS` – set of variables used when expanding Ginger PSL templates. For example, `GINGER_DEFINITIONS "foo=bar baz=quux USE_DYNLD"` sets the `foo` variable to `bar`, sets the `baz` variable to `quux`, and sets the `USE_DYNLD` variable to `TRUE`. The `USE_DYNLD` variable is set to `TRUE` because variables used when expanding Ginger PSL templates are set to `TRUE` by default.

Example call:

```
include (${KL_SDK_ROOT_PATH}/share/vendor_rom/build_vendor_rom.cmake)

build_vendor_romfs (vendor_romfs
                    SECURITY_PSL_FILE src/security.psl
                    EDL_TARGETS       Hello_edl_files)
```

kl_install_packages library

The `kl_install_packages` library is included in the KasperskyOS SDK and contains the `kl_install_packages()` command. This command is declared in the file named `/opt/KasperskyOS-Community-Edition-<platform>-<version>/toolchain/share/cmake/Modules/package_manager/kl_install_packages.cmake`.

```
kl_install_packages(IMAGE_NAME ...)
```

This command unpacks KPA packages into the root file system rootfs.

This lets you avoid copying the KPA package to rootfs and then installing it upon the first startup, using the ServiceManager system program and based on the initial_install.json description. Instead, you can unpack the package into the relevant directory inside rootfs while rootfs is being created.

Parameters:

- `OUTPUT_FILES_VAR` – name of the `CMake` variable that is created by the command. Required parameter.
- `PACKAGES_DIR` – you must specify `"${PACKAGES_BUILD_DIR}"`.
- `DB_PATH` – you must specify `"${ROOTFS_DIR}/data${MINIMAL_KOS_PACKAGE_MANAGER_DB_PATH}"`.
- `APPS_PATH` – you must specify `"${ROOTFS_DIR}/${MINIMAL_KOS_PACKAGES_INSTALLED_DIR}"`.

- `SCHEMA_PATH` – you must specify `"${ROOTFS_DIR}/${MINIMAL_KOS_PACKAGE_MANAGER_SCHEMA_DIR}"`.
- `PKG_EXTENSION` – you must specify `"kpa"`.
- `PACKAGES` – list of KPA packages for installation in the root file system rootfs.
- `DEPENDS <DEPENDING_TARGET1> [<DEPENDING_TARGET2>]...` – names of the CMake targets that must be executed before running this command.

Example call:

```
kl_install_packages (
    OUTPUT_FILES_VAR installed_packages
    PACKAGES_DIR "${PACKAGES_BUILD_DIR}"
    DB_PATH "${ROOTFS_DIR}/data${MINIMAL_KOS_PACKAGE_MANAGER_DB_PATH}"
    APPS_PATH "${ROOTFS_DIR}/${MINIMAL_KOS_PACKAGES_INSTALLED_DIR}"
    SCHEMA_PATH "${ROOTFS_DIR}/${MINIMAL_KOS_PACKAGE_MANAGER_SCHEMA_DIR}"
    PACKAGES "PreinstalledApp"
    PKG_EXTENSION "kpa"
    DEPENDS PreinstalledApp.kpa rootfs_part_src)
```

Using dynamic libraries

Advantages of using dynamic libraries

Using dynamic libraries (`*.so` files) in a solution provides the following advantages:

- Efficient use of RAM.

  Many executable files and other dynamic libraries in a solution can depend on the same dynamic library. In this case, the part of the dynamic library containing the code and read-only data is loaded into RAM in a single instance.

  Dynamic libraries can be loaded into and unloaded from RAM on the initiative of the programs that use them. This makes it possible to ensure that dynamic libraries are in RAM only while they are actually being used.

- Convenient software updates.

  A dynamic library update is applied to all executable files and dynamic libraries dependent on this dynamic library without having to rebuild them.

- Capability to implement a mechanism for plug-ins.

  Plug-ins for solution components consist of dynamic libraries.

- Shared use of code and data.

  The same dynamic library can be shared by multiple processes. In addition to conserving RAM, this eliminates the need to duplicate common program code (placed in the dynamic library) in the executable files and allows storage of common program data (read-only) in the dynamic library.

> We cannot guarantee the performance of any dynamic libraries outside of those supplied as part of the KasperskyOS SDK or those created by the solution developer.
> Currently, the dynamic libraries `libc.so` and `libpthread.so` are not provided as part of the KasperskyOS SDK.

Special considerations when building executable files using dynamic libraries

Executable files using dynamic libraries must be dynamically linked. (For details on CMake commands that specify how to link executable files, see platform library.)

> An executable file that must be linked to a dynamic library can be linked to a static library if the necessary dynamic library is not found. For example, if the CMake `target_link_libraries (client m)` command is being used, the executable file of the `client` program is linked to the static library `libm.a` if the dynamic library `libm.so` is not found.

Adding dynamic libraries to a solution

To add dynamic libraries to the solution, you need to place them in KPA packages along with those programs that depend on these dynamic libraries. To do so, you need to use the CMake command `add_kpa_component ()`. If the dynamic library is included in a KPA package, after its installation this library will be placed into the directory `/data/packages.unpacked/<KPA package identifier>/<KPA package version>/lib.` This path must be specified in calls of the `dlopen()` function of the POSIX interface and/or specified as the search path for dynamic libraries. Search paths for dynamic libraries can be specified via the `LD_LIBRARY_PATH` environment variable or in the `DT_RUNPATH` and `DT_RPATH` fields of the `.dynamic` section of ELF files (for more details, see Life cycle of a dynamic library). For example, this can be done by defining the value of the `LD_LIBRARY_PATH` environment variable in the `RUN_CONFIGURATION->ENV` parameter of the CMake command `add_kpa_component ()`:

```
add_kpa_component (${PACKAGE}
                   RUN_CONFIGURATION "default"
                   ...
                   # The KOS_APP_PACKAGE_DIR environment variable will
                   # receive the absolute path to the installation directory of the
                   # specified version of the KPA package when the program is started.
                   # The KPA package version is defined in the VERSION parameter
```

```
# of the CMake command add_kpa_package ().
ENV                "LD_LIBRARY_PATH=\${KOS_APP_PACKAGE_DIR}lib")
```

Dynamic libraries may depend on other dynamic libraries, so each dynamic library must be added with all dependencies.
It is not recommended to place dynamic libraries into the `/lib` directory because it contains system dynamic libraries.

Life cycle of a dynamic library

The life cycle of a dynamic library includes the following phases:

1. Loading into memory.

   The dynamic library is loaded from the file system into RAM by the BlobContainer system program that comes with KasperskyOS. (To gain access to file systems with dynamic libraries, the BlobContainer program uses IPC to query the VfsStorage system program that comes with KasperskyOS.) The part of a dynamic library that contains code and read-only data is placed into shared memory, which is mapped by the dynamic library loader onto the memory of all processes that use this library. (The dynamic library loader code is included in the dynamic library `ld.elf_so` that is linked to executable files, or in the executable files themselves, depending on the dynamic linking option being used. For more details, refer to the platform library section.) The other part containing read-and-write-accessible data is copied by the dynamic library loader into the memory of each of these processes. Loading is performed either upon starting a process created from an executable file linked to the dynamic library, or as a result of calling the `dlopen()` function of the POSIX interface. (If the required dynamic library is already loaded when the process is started or when the `dlopen()` function is called, it is not loaded again. Instead, the already loaded library is used.)

   A dynamic library can depend on other dynamic libraries, therefore the executable file depends not only on the dynamic library it directly uses but also on all direct and indirect dependencies of that library. A dynamic library is loaded together with all of the dynamic libraries that it depends on.

   If you use the `LD_PRELOAD` environment variable to specify a list of dynamic libraries for a program, these dynamic libraries will be loaded at program startup even if the executable file does not depend on them. (The list items must be absolute paths to dynamic libraries, separated by colons. For example: `LD_PRELOAD=/home/user/lib/libfoo.so:/usr/lib/libbar.so`.) The functions that are exported by the dynamic libraries specified in `LD_PRELOAD` replace the same-named functions that are exported by other dynamic libraries. This can be used for debugging purposes if you need to replace functions imported from dynamic libraries.

   If the outdated version of dynamic linking is used, it is impossible to use the `LD_PRELOAD` environment variable to replace functions of the `libkos`, `libc`, `libm`, `libpthread`, `libvfs`, `librtlcom`, and `libdynld` libraries.

   The search for dynamic libraries in file systems is performed in the following order:

   1. Absolute paths defined through the `LD_LIBRARY_PATH` environment variable.

      Paths must be separated by a colon, for example: `LD_LIBRARY_PATH=/usr/lib:/home/user/lib`.

2. Absolute paths stored in the `DT_RUNPATH` or `DT_RPATH` field of the `.dynamic` section of executable files and dynamic libraries.

   Paths must be separated by colons. They can be saved in executable files linked to dynamic libraries and in dynamic libraries linked to other dynamic libraries. To do so, for example, you can use the CMake command `set_target_properties ()` while specifying the paths in the `INSTALL_RPATH` parameter. The paths are used when searching for the required dynamic libraries (specified in the `DT_NEEDED` fields of the `.dynamic` section) only for the specific executable file (or dynamic library) in which they are saved.

3. Path `/lib`.

If a relative path to a dynamic library is specified through the `file` parameter of the `dlopen()` function, the search for this library and the libraries on which it depends is performed in the same order.

2. Use by a process (or processes).

3. Unloading from memory.

   A dynamic library that is no longer required is removed from RAM. This occurs when all processes using this library have terminated or have called the `dlclose()` function of the POSIX interface (the number of calls of the `dlclose()` function must be at least the number of `dlopen()` function calls). A dynamic library linked to an executable file cannot be deleted from memory until termination of the process that was created based on this executable file. A dynamic library that is linked to other dynamic libraries is deleted after all libraries that depend on it are deleted.

Building dynamic libraries

To build a dynamic library, you need to use the following CMake command:

```
add_library (<library build target name> SHARED <path to a file of the library source code>…)
```

You can also build a dynamic library by using the following CMake command:

```
add_library (<library build target name> <path to a file of the library source code>…)
```

In this case, the CMake variable `BUILD_SHARED_LIBS` must have the "true" value. Otherwise, a static library will be built.

By default, the library file name matches the name of the build target defined via the parameter of the CMake `add_library()` command. The library file name can be changed by using the CMake `set_target_properties ()` command. This can be done to make the library file name identical for its dynamic and static variants.

Example:

```
# Build the static library
add_library (somelib_static STATIC src/somesrc.cpp)
set_target_properties (somelib_static PROPERTIES OUTPUT_NAME "somelib")
# Build the dynamic library
add_library (somelib_shared SHARED src/somesrc.cpp)
set_target_properties (somelib_shared PROPERTIES OUTPUT_NAME "somelib")
```

A dynamic library can be linked to other static and dynamic libraries by using the CMake `target_link_libraries ()` command. In this case, static libraries must be built with the `-fPIC` flag. This flag is applied when building a static library if the following CMake command is used:

```
set_property (TARGET <library build target name>… PROPERTY POSITION_INDEPENDENT_CODE ON)
```

Cross compilers

The toolchain provided in the KasperskyOS SDK includes one or more Clang compilers and a rustc compiler. The `toolchain/bin` directory contains the following files:

- Executable files of compilers (for example, `clang-17` or `rustc`)
- Executable files of linkers (for example, `*-kos-ld`)
- Executable files of assemblers (for example, `*-kos-as`)
- Bash scripts for the Clang compiler (for example, `*-kos-clang`, `*-kos-clang++`)

Debugging programs in a KasperskyOS-based solution

Using the GDB debugger

To debug programs in a KasperskyOS-based solution, you must use the GDB debugger from the KasperskyOS Community Edition SDK on a computer running a Linux OS. In this case, you need to configure the connection to the GDB server of the KasperskyOS kernel. Debugging can be performed in the QEMU from KasperskyOS Community Edition SDK or on Arm64 hardware platform. For RISC-V architecture, debugging is supported only via JTAG or OpenOCD.

Stack backtrace when a process abnormally terminates

If a process terminates unexpectedly, the KasperskyOS kernel prints stack backtrace data (call stack information) for the thread in which the unhandled exception occurred. This data lets you determine the sequence of function calls that led to the unexpected termination of the process.

> In some cases of abnormal process termination, the KasperskyOS kernel cannot prepare information about the call stack. For instance, this could be caused by optimization parameters that were applied when building a program that was run in the context of the unexpectedly terminated process.

Support for sanitizers

KasperskyOS Community Edition SDK supports the ASAN and UBSAN sanitizers.

KasperskyOS includes a handle sanitizer.

Using the GDB debugger

> Correct operation of the GDB debugger and QEMU provided by KasperskyOS Community Edition SDK cannot be guaranteed.
> By default, the GDB debugger supports OS ABI (Operating System ABI), which is specific to KasperskyOS. If the default OS ABI profile is modified, the capability for full-fledged debugging cannot be guaranteed.

You can debug multiple programs of a solution at the same time by loading debug symbols for multiple executable files. Debugging on hardware platforms with SMP (including for emulation in QEMU) is also supported.

Initial steps of debugging in QEMU

To begin debugging using the GDB server of the KasperskyOS kernel in QEMU, you must complete the following steps:

1. Build debug versions of executable files and libraries, create a solution image for QEMU, and run QEMU.

   To do so, call the `cmake` shell commands, and specify the `-D CMAKE_BUILD_TYPE=Debug` and `--target gdbsim` parameters.

   Example:

   ```
   $ cmake -B build \
           -D CMAKE_TOOLCHAIN_FILE="$KOSCEDIR/toolchain/share/toolchain-*-kos.cmake" \
           -D CMAKE_BUILD_TYPE=Debug \
           && cmake --build build --target gdbsim
   ```

   Instead of the `gdbsim` target, you can specify the `gdbsim/fast` target to avoid rebuilding.

   QEMU starts, but solution programs do not start running while the kernel GDB server is waiting for the GDB debugger to connect. This wait continues indefinitely.

   Debug symbols can be saved both in ELF files and in separate files. In the latter case, you can significantly reduce the size of the solution image. To save debug symbols in `*.dbg` files, use the following `CMake` command:

   ```
   add_custom_command(TARGET <build target name> POST_BUILD
                      COMMAND ${CMAKE_OBJCOPY} --only-keep-debug $<TARGET_FILE:<build target name>>
   $<TARGET_FILE_NAME:<build target name>>.dbg
                      COMMAND ${CMAKE_OBJCOPY} --strip-debug $<TARGET_FILE:<build target name>>
                      COMMAND ${CMAKE_OBJCOPY} --add-gnu-debuglink=$<TARGET_FILE_NAME:<build target name>>.dbg
   $<TARGET_FILE:<build target name>>)
   ```

2. Run the GDB debugger and connect to the GDB server of the kernel.

   To do so, you need to call the `cmake` shell command by specifying the `--target gdb/fast` parameter.

   Example:

   ```
   $ cmake --build build --target gdb/fast
   ```

   The GDB debugger will execute commands from the automatically generated `build/einit/.gdbinit` file.

3. Create inferiors, attach them to processes, and load debug symbols.

   To load debug symbols, you need to switch to the inferior associated with the process and use the following GDB commands:

   - `add-symbol-file <path to file>` – for executable files with static or dynamic linking.
   - `file <path to file>` – for executable files with dynamic linking.

   The GDB command `file` must be used so that the GDB debugger can load debug symbols of an executable file and the dynamic libraries required by this executable file. Use the GDB command `add-symbol-file` for the GDB debugger to load the debug symbols only for an executable file with dynamic linking.

If debug symbols are saved in separate files instead of in the executable files, links to the files containing the debug symbols are added to the executable files. When the GDB command `add-symbol-file` or `file` is called, you can specify the executable file or the file containing debug symbols.

> When performing a repeat build (step 1), you need to terminate the debug session (exit the GDB debugger) and repeat steps 2 and 3.

Preparations for debugging on the hardware platform

To prepare for debugging using the GDB server of the KasperskyOS kernel on the hardware platform, you must complete the following steps:

1. Establish a switched connection between the hardware platform and the computer where the GDB debugger will operate.

   You must use USB-UART converters to connect the computer to the hardware platform. The diagrams for connecting USB-UART converters and hardware platforms are provided in the Preparing Raspberry Pi 4 B to run examples and Preparing OrangePi RV2 to run examples sections.

2. Define the kernel startup parameters:

   - `gdbstub`
   - `gdbstub.transport=uart:<UART ID>[?baudrate=<baud rate>]`
   - [optional] `gdbstub.connection.timeout=<parameter value>`
   - [optional] `earlycon={<UART description>|none}`

3. Create a `.gdbinit` file so that you do not need to manually call the initial GDB commands every time the GDB debugger is started. (The `.gdbinit` file must be saved in the directory where the GDB debugger will be run.)

   Example:

```
# Define a path to search for dynamic libraries containing debug symbols from
# KasperskyOS Community Edition SDK
set sysroot /opt/KasperskyOS-Community-Edition-<platform>-<version>/sysroot-*-kos
# Define additional paths to search for dynamic libraries containing debug
# symbols
set solib-search-path /home/user/example/build/lib1:/home/user/example/build/lib2
# Define the symbol rate for interaction with the GDB server of the kernel
# You must set the value to 115200 baud.
set serial baud 115200
# Connect the GDB debugger to the GDB server of the kernel
# You must define the USB port that is used to connect the USB-UART converter
# used for debugging. You must also specify that the GDB server of the kernel
# interacts with the GDB debugger via an extended protocol.
target extended-remote /dev/ttyUSB1
# Configure thread management (optional)
# The GDB debugger can operate in all-stop or non-stop mode.In the first mode,
# when the GDB debugger sees the stoppage of the thread whose context
# is in the debug focus, it assumes that all other threads
# known to the GDB debugger also stop. In the second mode, the GDB debugger assumes that
# all threads except the stopped thread continue to run. All-stop mode
# applied by the GDB debugger by default, and the GDB server of the kernel provides the capability to use
# only this mode. To edit the default settings
# for managing threads in all-stop mode, use GDB commands
# set scheduler-locking and set schedule-multiple.
# In this example, the first GDB command indicates that, when resuming
# execution of a thread whose context is in the debug focus (for example,
# the following GDB commands: continue, step, next), the GDB debugger must resume execution of all other
threads.
# The second GDB command indicates that the GDB debugger, when resuming execution of a thread
# whose context is in the debugging focus, must resume execution of all
# threads included in the processes of all programs that are being debugged.
set scheduler-locking off
set schedule-multiple on
```

Initial steps of debugging on the hardware platform

To begin debugging using the GDB server of the KasperskyOS kernel on the hardware platform, you must complete the following steps:

1. Build debug versions of executable files and libraries and create a solution image for the hardware platform.

   To do so, call the `cmake` shell commands, and specify the `--D CMAKE_BUILD_TYPE=Debug` and `--target sd-image` parameters.

   Example:

```
$ cmake -B build \
        -D CMAKE_TOOLCHAIN_FILE="$KOSCEDIR/toolchain/share/toolchain-*-kos.cmake" \
        -D CMAKE_BUILD_TYPE=Debug \
        && cmake --build build --target sd-image
```

   Debug symbols can be saved both in ELF files and in separate files. In the latter case, you can significantly reduce the size of the solution image. To save debug symbols in `*.dbg` files, use the following `CMake` command:

```
add_custom_command(TARGET <build target name> POST_BUILD
                   COMMAND ${CMAKE_OBJCOPY} --only-keep-debug $<TARGET_FILE:<build target name>>
$<TARGET_FILE_NAME:<build target name>>.dbg
                   COMMAND ${CMAKE_OBJCOPY} --strip-debug $<TARGET_FILE:<build target name>>
                   COMMAND ${CMAKE_OBJCOPY} --add-gnu-debuglink=$<TARGET_FILE_NAME:<build target name>>.dbg
$<TARGET_FILE:<build target name>>)
```

2. Load the solution image to the hardware platform and start the solution.

   Solution programs do not start running while the kernel GDB server is waiting for the GDB debugger to connect.

3. Run the GDB debugger and connect to the GDB server of the kernel.

   To do so, run the executable file `toolchain/bin/*-kos-gdb` from KasperskyOS Community Edition SDK. It must be run in the directory where the `.gdbinit` file created during preparation for debugging is saved.

4. Create inferiors, attach them to processes, and load debug symbols.

   To load debug symbols, you need to switch to the inferior associated with the process and use the following GDB commands:

   - `add-symbol-file <path to file>` – for executable files with static or dynamic linking.
   - `file <path to file>` – for executable files with dynamic linking.

   The GDB command `file` must be used so that the GDB debugger can load debug symbols of an executable file and the dynamic libraries required by this executable file. Use the GDB command `add-symbol-file` for the GDB debugger to load the debug symbols only for an executable file with dynamic linking.

   If debug symbols are saved in separate files instead of in the executable files, links to the files containing the debug symbols are added to the executable files. When the GDB command `add-symbol-file` or `file` is called, you can specify the executable file or the file containing debug symbols.

> When performing a repeat build (step 1), you need to terminate the debug session (exit the GDB debugger) and repeat steps 2–4.

Working with processes and threads

In the GDB debugger, objects called *inferiors* are used to work with processes. The process of each program to be debugged must be attached to a separate inferior. Switching between inferiors is equivalent to switching the debug focus to the context of another process. The debug symbols for the program to be debugged must be loaded after switching to the inferior that is attached to the process of this program.

An inferior is automatically created when the GDB debugger is started. When the GDB debugger connects to the GDB server of the kernel, execution of the solution code stops and this inferior is automatically attached to the process containing the thread in whose context the solution code execution is stopped. All other required inferiors can be created, attached to processes, and detached from processes manually or automatically. In the first case, you need to use the GDB commands for working with inferiors. In the second case, you must configure handling of events in the process life cycle.

GDB commands for working with inferiors

To work with inferiors, you need to use the following GDB commands:

- `info inferiors`

  Displays the list of inferiors. The current inferior is marked by the `*` character. (The context of the process associated with the current inferior is in the debug focus.)

- `add-inferior`

  Creates an inferior.

- `inferior <inferior number>`

  Switches to another inferior.

- `attach <PID>`

  Attaches the current inferior to the process with the specified PID. To print a list of processes, run the GDB command `info os processes`.

- `detach`

  Detaches the current inferior from the process.

Configuring handling of events in the process life cycle

The GDB server of the kernel notifies the GDB debugger about the occurrence of the following events:

1. Creation of a child process
2. Start of the initial thread of a child process
3. Termination of a process

The GDB debugger handles event 1 only if an inferior is attached to the parent process. To handle event 2, the GDB debugger must have previously handled event 1 when the child process was created. In this case, event 2 is handled even if the parent process was already detached from the inferior, for example, due to its termination.

To configure handling of events 1 and 2, use the following GDB commands:

- `set follow-fork-mode {child|parent}`

  Instructs the GDB debugger to switch to the context of the child process (`child`) or to remain in the context of the parent process (`parent`) when event 1 occurs. The second option is carried out by default.

- `set detach-on-fork {on|off}`

  Instructs the GDB debugger to leave the parent or child process attached to the inferior (`on`) or to leave the parent process attached to the inferior and create a new inferior and attach it to the child process (`off`) when event 1 occurs. The first option is carried out by default. In this case, the parameter of the GDB command `set follow-fork-mode` determines which process is attached to the inferior (parent or child).

- `set follow-exec-mode {new|same}`

  Instructs the GDB debugger to create a new inferior and attach it to the child process (`new`) or refrain from doing so (`same`) when event 2 occurs. The second option is carried out by default.

It is recommended to configure handling of events 1 and 2 as follows:

```
set follow-fork-mode parent
set follow-exec-mode same
set detach-on-fork off
```

This configuration ensures that events 1 and 2 are handled by the GDB debugger as follows:

- When event 1 occurs, a new inferior is created and attached to the child process.
- When event 1 occurs, the parent process remains attached to the inferior.
- When event 2 occurs, a new inferior is not created for the child process.

To avoid using an infinite loop, handling of event 2 by the GDB debugger can be used to stop execution of the program before management is transferred to the `main()` function.

To stop execution of the program when event 2 occurs, use the following GDB command:

```
catch exec
```

This GDB command stops the execution of each child process that was created while event 1 was handled. To stop the execution of individual processes, use the following GDB command:

```
monitor exec [<name of process 1>][, <name of process 2>]...
```

A subsequent call of this GDB command cancels the previous call, and calling this command without parameters will stop each child process that was created while event 1 was handled.

Event 3 is handled by the GDB debugger only if inferiors attached to terminated processes are created. When this event is handled, the GDB debugger detaches the inferior from the terminated process. Handling of event 3 does not need to be configured.

GDB commands for working with threads

To print a list of the threads contained in processes attached to inferiors, run the following GDB command:

```
info threads
```

The current thread is marked by the `*` character. (The context of the current thread is in the debug focus.)

To switch to the context of another thread, use the following GDB command:

```
thread <Id value received by calling the GDB command info threads>
```

Switching to the context of another thread will result in switching to another inferior if this thread is in a process attached to an inferior different from the current one.

Analyzing stack backtrace data when a process abnormally terminates

When a process is terminated due to an unhandled exception, the KasperskyOS kernel prints stack backtrace data in the following format:

```
Call Trace (User Mode):
    [<0000000004068498>] (__assert_func+0x78) from [<000000000405f478>] (_vfs_backend_create+0xe8)
    [<000000000405f478>] (_vfs_backend_create+0xe8) from [<00000000040bdab0>] (_vfs_init+0xd0)
    [<00000000040bdab0>] (_vfs_init+0xd0) from [<00000000040111a8>] (init_vfs_helper+0xc8)
    [<00000000040111a8>] (init_vfs_helper+0xc8) from [<00000000040117e0>] (__eprol+0x88)
    [<00000000040117e0>] (__eprol+0x88)
```

Here you can clearly see the sequence of function calls that led to the unexpected termination of the process. At the end of the sequence, the `_vfs_backend_create()` function code residing in the process memory at the address `0x000000000405f478` called the `__assert_func()` function, and the `__assert_func()` function code residing in the process memory at the address `0x0000000004068498` invoked the exception. In other words, the exception occurred during verification of the conditions in the `_vfs_backend_create()` function. (An entry in the format `<function name>+0x<offset>` indicates an address within the function code. For example, the entry `_vfs_backend_create+0xe8` indicates an address within the `_vfs_backend_create()` function code offset by `0xe8` bytes relative to the address of the start of this function's code. The entry `_vfs_backend_create+0xe8` corresponds to the address `0x000000000405f478`.)

The names of all or some functions in the sequence of calls may fail to appear, which makes it more difficult to determine the cause of the abnormal process termination.

Example stack backtrace data containing only memory addresses:

```
Call Trace (User Mode):
    [<0000000004068498>] from [<000000000405f478>]
    [<000000000405f478>] from [<00000000040bdab0>]
    [<00000000040bdab0>] from [<00000000040111a8>]
    [<00000000040111a8>] from [<00000000040117e0>]
    [<00000000040117e0>]
```

The reason for the missing names of functions in the stack backtrace data is as follows. To print stack backtrace data, the kernel gets the names of functions from the symbol table `.symtab` and the string table `.strtab` that are loaded into the memory of the abnormally terminated process from the debug version of the executable file. These tables may be incomplete or entirely missing from the memory of the abnormally terminated process. In addition, the kernel cannot print the names of functions that were imported from dynamic libraries.

The symbol table `.symtab` and string table `.strtab` are absent from process memory in the following cases:

- The process was created from the normal release version of the executable file. (The executable file was built as the normal release version, and the normal release versions of static libraries were included in it during the build.)

  In contrast to the debug version, the normal release version of the executable file does not contain the symbol table `.symtab` and string table `.strtab`.

- The process was created from the debug version of an executable file from which the symbol table `.symtab` and string table `.strtab` were deleted before the file was added to the solution image. For instance, they could have been deleted by the `strip` utility.

- The process was created from the debug version of an executable file from which the symbol table `.symtab` and string table `.strtab` were not deleted before the file was added to the solution image, but these tables were not loaded into the process memory.

  The high-level API for process management and the low-level API for process management let you create a process with optional loading of the symbol table `.symtab` and string table `.strtab`.

The symbol table `.symtab` and string table `.strtab` are incomplete in the following cases:

- If the executable file was built as a debug version, the normal release versions of static libraries were included in the file during the build, and the symbol table `.symtab` and string table `.strtab` will not contain information about the functions implemented in these libraries.

- If the executable file was built as a normal release version, the debug versions of static libraries were included in the file during the build, and the symbol table `.symtab` and string table `.strtab` will contain information only about the functions that are implemented in these libraries.

For some memory addresses specified in stack backtrace data, information about functions can be obtained from the executable file. If a memory address is present in the symbol table `.symtab` of the executable file, you can get the code line number and the name of the function corresponding to this memory address. If a memory address is missing from the symbol table `.symtab` but is present in the symbol table `.dynsym` of the executable file, you can get the name of the function corresponding to this memory address. (When using ASLR, the memory addresses specified in the stack backtrace data are increased by the `Relocation base` value for the ELF image load offset that is printed together with the stack backtrace data.) Otherwise, information about functions cannot be received from the executable file. For instance, this applies to functions imported from dynamic libraries and to functions implemented in normal release versions of static libraries.

An executable file in KasperskyOS contains a set of exportable functions (but is not a dynamic library). For example, the functions of the `libc` library included in executable files are exportable. The symbol table `.dynsym` contains information about exportable functions. This table is present in the release version of the executable file, and remains in the debug version of the executable file after the debug symbols are deleted from it. (The symbol table `.symtab` and string table `.strtab` are part of the debug symbols.)

To get information about functions corresponding to the memory addresses specified in backtrace data from an executable file, use the `addr2line` utility provided in KasperskyOS Community Edition SDK (the `toolchain/bin/*-kos-addr2line` executable file). Format of the command for starting the `addr2line` utility:

```
*-kos-addr2line [--functions] --exe=<path to the executable file> 0x<address 1> [0x<address 2>]...
```

In this command, you need to specify the memory addresses from the stack backtrace data reduced by the `Relocation base` value. The `--functions` parameter requires output of the names of functions in addition to the numbers of source code lines.

Example:

```
$ *-kos-addr2line --functions --exe=./build/hello/Hello 0x30090
main
/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples/hello/hello/src/hello.c:6
```

Using a handle sanitizer

A handle sanitizer lets you detect the following problems:

- Use of invalid handles (closed and uncreated)
- Leaked handles (presence of unclosed handles that are no longer required)

The handle sanitizer must be activated separately in each process for which checks for invalid handles and/or handle leaks are required. To activate the handle sanitizer, you need to pass the `KOS_HANDLE_SANITIZER` environment variable with the value `True` to the process.

Example of passing the `KOS_HANDLE_SANITIZER` environment variable to a process when packing a KPA package using the environment variable in the `RUN_CONFIGURATION->ENV` parameter of the CMake command add_kpa_component ():

```
add_kpa_component (${PACKAGE}
                   RUN_CONFIGURATION "default"
                   ...
                   ENV               "KOS_HANDLE_SANITIZER=True")
```

Messages from the handle sanitizer about detected problems are relayed to the logging system, which prints diagnostic data via the UART or saves it to a file, for example. For each detected problem, this information contains a brief description of the problem, the value of the handle, and the stack backtrace data, which includes information about the sequence of function calls that resulted in completed operations from the following set: create a handle, close a handle, or use an invalid handle. The specific operations from this set that are reflected in the stack backtrace data depend on the problem that was detected.

Analyzing stack backtrace data

Stack backtrace data looks as follows:

```
...
[2025-12-02T07:49:39.334][Error][Hello][22:22][HSAN]    0xbab42e80363 <KnHandleConnectEx+0xbab42cbc073> at helloe
[relocBase] 0xbab42cbc000 [buildId] da0a1724ad9f32798ee18c46df9340d7e1bf2c5a
[2025-12-02T07:49:39.337][Error][Hello][22:22][HSAN]    0xbab42e803e3 <KnHandleConnect+0xbab42cbc043> at helloe
[relocBase] 0xbab42cbc000 [buildId] da0a1724ad9f32798ee18c46df9340d7e1bf2c5a
[2025-12-02T07:49:39.341][Error][Hello][22:22][HSAN]    0xbab42e80c3e <KnHandleCreateListener+0xbab42cbc04e> at
helloe [relocBase] 0xbab42cbc000 [buildId] da0a1724ad9f32798ee18c46df9340d7e1bf2c5a
...
```

Here, the `KnHandleCreateListener()` function code located in the process memory at the address `0xbab42e80c3e` called the `KnHandleConnect()` function, and the `KnHandleConnect()` function code located in the process memory at the address `0xbab42e803e3` called the `KnHandleConnectEx()` function. (An entry in the format `<function name>+0x<offset>` means the address inside the function code. For example, the entry `KnHandleCreateListener+0xbab42cbc04e` indicates an address within the `KnHandleCreateListener()` function code offset by `0xbab42cbc04e` bytes relative to the address of the start of this function's code. The `KnHandleCreateListener+0xbab42cbc04e` entry corresponds to the address `0xbab42e80c3e`.) As a result of using ASLR, the memory addresses are increased by the ELF image load offset value (`[relocBase] 0xbab42cbc000`).

> The stack backtrace data will not contain the names of functions whose information is not available in the `.symtab` symbol table and the `.strtab` table of strings loaded into the process memory. For more details, refer to Analyzing stack backtrace data when a process abnormally terminates.

Detecting the use of invalid handles

Invalid handle usage is detected automatically in all processes in which the handle sanitizer is enabled. If the use of a closed handle is detected, the stack backtrace data includes information about the sequence of function calls that resulted in the handle being created, closed, and used after it was closed. If the use of an uncreated handle is detected, the stack backtrace data includes information about the sequence of function calls that resulted in the use of the uncreated handle.

> The functionality for detecting the use of invalid handles can be disabled by `setting the kernel startup parameter` handle.disable_detect_invalid_access.

Detecting handle leakage

Handle leaks are detected by an API that is defined in the header file `sysroot-*-kos/include/hsan/leak_control.h` from the KasperskyOS SDK.

The API lets you detect leaks of handles both in the process calling API functions and in processes that are external to this process. In the first case, you need to activate the handle sanitizer in the process that calls the API functions. In the second case, you need to activate the handle sanitizer only in external processes. Also, in the second case, you need to statically create IPC channels from the process that calls the API functions and from external processes to the DCM system program process.

Checking external processes for handle leaks may be required if these processes are in any way connected to the process that calls the API functions. For example, if the process that calls the API functions interacts with external processes via IPC, and the process that calls the API functions is the client process and the external processes are server processes, it may be necessary to verify that all handles created at a certain stage of execution of the program in the client process are closed in the server processes.

Information about API functions is provided in the table below.

`HandleLeakControlExt*()` functions are intended to detect leaks of handles in external processes.

The standard scenario for API usage includes the following steps:

1. [Optional] Create a set of external processes for which you need to check for leaks of handles.

   This step must be performed if you need to check external processes.

   To add a process to the set, call the `HandleLeakControlExtAddTask()` function.

2. Start tracking the creation and closing of handles by calling the `HandleLeakControlStart()` or `HandleLeakControlExtStart()` function.

   Only those handles that were generated after this step and before completion of step 4 are monitored.

3. Check for any unclosed monitored handles.

   To complete this step, you must use the `HandleLeakControlCheck*()` or `HandleLeakControlExtCheck*()` functions.

   The `HandleLeakControlCheck()` function uses the `leakDetected` output parameter to pass a boolean value that reflects whether unclosed monitored handles were detected in the calling process when this function was called. If unclosed monitored handles were detected, messages about the handle leakage are relayed to the logging system. The backtrace data includes information about the sequences of function calls that generated the unclosed monitored handles.

   The `HandleLeakControlCheckNoPrint()` function differs from the `HandleLeakControlCheck()` function only in that the messages about leaked handles are not forwarded in the logging system.

   The `HandleLeakControlCheckTimeouted()` function checks for the presence of unclosed monitored handles in the calling process at the time of this function call and, if detected, does not return control immediately but starts a cyclic check while waiting for the closure of monitored handles for no longer than the time specified in the `timeoutMs` parameter. The `HandleLeakControlCheckTimeouted()` function uses the `leakDetected` output parameter to pass a boolean value that reflects whether unclosed monitored handles were detected in the calling process when exiting the function. If unclosed monitored handles are detected when exiting the function, messages containing the same information as when using the `HandleLeakControlCheck()` function are relayed to the logging system.

   The `HandleLeakControlExtCheck*()` functions differ from the `HandleLeakControlCheck*()` functions only in that they perform the check not in the calling process but in one or more processes external to the caller.

4. Stop tracking the creation and closing of handles by calling the `HandleLeakControlStop()` or `HandleLeakControlExtStop()` function.

5. [Optional] Remove processes from the set of external processes for which a handle leak check was performed.

   This step must be performed if external processes were checked.

   To remove a process from the set, call the `HandleLeakControlExtRemoveTask()` function.

leak_control.h functions

Function	Information about the function
`HandleLeakControlStart()`	Purpose Starts tracking the creation and closing of handles in the calling process. Parameters N/A Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`HandleLeakControlCheck()`	Purpose Checks for the presence of unclosed monitored handles in the calling process at the time of the call. Messages about handle leaks are relayed to the logging system. Parameters • [out] `leakDetected` – pointer to a boolean value reflecting whether unclosed monitored handles were detected (`rtl_true` means yes, `rtl_false` means no). Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`HandleLeakControlCheckNoPrint()`	Purpose Checks for the presence of unclosed monitored handles in the calling process at the time of the call. Messages about handle leaks are not relayed to the logging system. Parameters • [out] `leakDetected` – pointer to a boolean value reflecting whether unclosed monitored handles were detected (`rtl_true` means yes, `rtl_false` means no). Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`HandleLeakControlCheckTimeouted()`	Purpose Checks for the presence of unclosed monitored handles in the calling process at the time of the call and, if detected, performs a cyclic check while waiting for the closure of monitored handles for no longer than the specified time. Messages about handle leaks are relayed to the logging system. Parameters • [in] `timeoutMs` – maximum timeout for the closure of monitored handles (in milliseconds). • [out] `leakDetected` – pointer to a boolean value reflecting whether unclosed monitored handles were detected when exiting the function (`rtl_true` means yes, `rtl_false` means no). Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information A non-blocking call if the `timeoutMs` parameter is set to `0`, or there are no unclosed monitored handles at the time of the call.
`HandleLeakControlStop()`	Purpose Stops tracking the creation and closing of handles in the calling process. Parameters N/A Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`HandleLeakControlExtAddTask()`	Purpose Adds a process to the set of external processes for which you need to perform a handle leak check. Parameters • [in] `taskName` – pointer to the external process name. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`HandleLeakControlExtRemoveTask()`	Purpose Removes a process from the set of external processes for which you need to perform a handle leak check. Parameters • [in] `taskName` – pointer to the external process name. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`HandleLeakControlExtStart()`	Purpose Starts tracking the creation and closing of handles in external processes. Parameters N/A Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`HandleLeakControlExtCheck()`	Purpose Checks for unclosed monitored handles in external processes at the time of the call. Messages about handle leaks are relayed to the logging system. Parameters • [out] `leakDetected` – pointer to a boolean value reflecting whether unclosed monitored handles were detected (`rtl_true` means yes, `rtl_false` means no). Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`HandleLeakControlExtCheckNoPrint()`	Purpose Checks for unclosed monitored handles in external processes at the time of the call. Messages about handle leaks are not relayed to the logging system. Parameters • [out] `leakDetected` – pointer to a boolean value reflecting whether unclosed monitored handles were detected (`rtl_true` means yes, `rtl_false` means no). Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Function	Information about the function
`HandleLeakControlExtCheckTimeouted()`	Purpose Checks for the presence of unclosed monitored handles in external processes at the time of the call and, if detected, performs a cyclic check while waiting for the closure of monitored handles for no longer than the specified time. Messages about handle leaks are relayed to the logging system. Parameters • [in] `timeoutMs` – maximum timeout for the closure of monitored handles (in milliseconds). • [out] `leakDetected` – pointer to a boolean value reflecting whether unclosed monitored handles were detected when exiting the function (`rtl_true` means yes, `rtl_false` means no). Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information A non-blocking call if the `timeoutMs` parameter is set to `0`, or there are no unclosed monitored handles at the time of the call.
`HandleLeakControlExtStop()`	Purpose Stops tracking the creation and closing of handles in external processes. Parameters N/A Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Using auxiliary tools for processing diagnostic data

KasperskyOS Community Edition SDK provides the `toolchain/bin/parse_hsan_dump.py` script, which performs the following operations:

- Selects handle sanitizer messages from diagnostic data.
- [Optional] Converts stack backtrace data, namely by replacing entries in the format `<function name>+0x<offset>` with entries in the format `<function name> at <absolute path to the source code file>:<string number>:<string indentation>` by using the `toolchain/bin/llvm-symbolizer` tool from KasperskyOS Community Edition SDK.

  To perform this conversion, you need to create ELF files (executable files, dynamic libraries) that contain debug data, or ELF files that do not contain debug data but are associated with special files in which debug data is saved separately. (For files containing debug data, see "Debugging programs in a KasperskyOS-based solution".)

  After conversion, the stack backtrace data contains the addresses of function calls reduced by the ELF image load offset value. (This offset results from the use of ASLR.)

Syntax of the shell command for running the `parse_hsan_dump.py` script:

```
parse_hsan_dump.py [-s <PATH_TO_LLVM-SYMBOLIZER> -b <PATH_TO_ELF_FILES>]
```

Parameters:

- `-s <PATH_TO_LLVM-SYMBOLIZER>` – absolute path to the executable file of the `llvm-symbolizer` tool.
- `-b <PATH_TO_ELF_FILES>` – path to the directory containing executable files and dynamic libraries.

  In this parameter, you need to specify the path to the directory containing the ELF files that have debug data (or that are associated with special files containing debug data). ELF files can be located in directories that are embedded into the specified one. For example, you can specify the path to the `build` directory from the solution CMake project.

  You can specify an absolute path or a path relative to the current directory.

The usage scenario for the `parse_hsan_dump.py` script includes the following steps:

1. Save diagnostic data to a file.

   Example:

   ```
   cmake --build build --target sim | tee log.txt
   ```

2. Send diagnostic data to the `parse_hsan_dump.py` script via the standard input stream.

   Example:

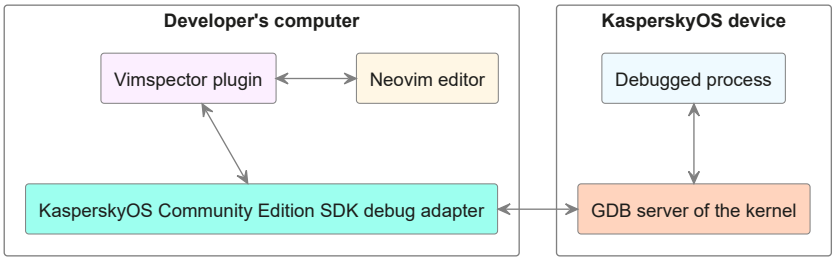
   ```
   cat log.txt | /opt/KasperskyOS-Community-Edition-<platform>-<version>/toolchain/bin/parse_hsan_dump.py -s
   /opt/KasperskyOS-Community-Edition-<platform>-<version>/toolchain/bin/llvm-symbolizer -b
   home/user/project/build
   ```

Debug adapter for the Neovim editor

KasperskyOS Community Edition SDK provides a debug adapter that is intended for use in the Neovim source code editor when debugging programs for KasperskyOS. The executable file of `debug_adapter` adapter is located in the `/opt/KasperskyOS-Community-Edition-<platform>-<version>/toolchain/bin/` directory.

> The debug adapter is included only in the deb package for running in the QEMU emulator.

It is assumed that the Vimspector plug-in will be used for debugging. The adapter can work with other debugging plug-ins (for example, nvim-dap), though this has not been tested.



Debug diagram

> Only "attach configuration: connect to process" is supported.

Debugging an application

If the KasperskyOS kernel is waiting for the debugger to connect before starting and you need to debug the process that is started from the KPA package, you can debug from startup.

Also see the training scenario Debugging an example in Neovim.

To debug an application, do the following:

1. Add to the project a `.vimspector.json` configuration file with the following contents for debugging in GDB:

```
.vimspector.json
```

```
{
  "adapters": {
    "kos-gdb": {
      "name": "kos-gdb",
      "command": [
        "/opt/KasperskyOS-Community-Edition-Qemu-<version>/toolchain/bin/debug_adapter",
        "--gdb-path",
        "/opt/KasperskyOS-Community-Edition-Qemu-<version>/toolchain/bin/aarch64-kos-gdb",
        "--transport",
        "stdio"
      ]
    }
  },

  "configurations": {
    "KOS attach (Hello)": {
      "default": true,
      "adapter": "kos-gdb",
      "filetypes": [ "c", "cpp" ],
      "configuration": {
        "request": "attach",
        "name": "Attach and debug",
        "gdbServer": "tcp::12345",
        "useExtendedRemote": true,
        "setupCommands": [
          {
            "description": "Load gdb_init",
            "text": "source  <path to project>/gdb_init",
            "ignoreFailures": false
          }
        ],
        "setupServerCommands": [
            {
            "description": "Load setup script",
            "text": "source <path to project>/setup",
            "ignoreFailures": false
          }
        ]
      }
    }
  }
}
```

   When using the example above, replace the `<version>` and `<path to project>` parameters with the SDK version and the path to the root directory of the example, respectively.

   For a description of the parameters used in the `.vimspector.json` file, see Adapter reference information.

2. Add the `gdb_init` file to the project:

```
gdb_init
```

```
shell rm -rf <path to project>/build/einit/sysroot-symbols/
shell mkdir <path to project>/build/einit/sysroot-symbols/
shell find -L <path to project>/build -type f -executable | xargs -I {} cp {} <path to
project>/build/einit/sysroot-symbols/
shell find -L <path to project>/build -type f -executable | xargs -I {} /opt/KasperskyOS-Community-Edition-
Qemu-<version>/toolchain/bin/cmake -DBINARY={} -DBUILD_ID_DIR=<path to project>/build/sysroot-symbols/ -P
/opt/KasperskyOS-Community-Edition-Qemu-<version>/toolchain/share/cmake/Modules/platform/building/build-
id.cmake 2> /dev/null
```

   Replace the `<version>` and `<path to project>` parameters with the SDK version and the path to the root directory of the example, respectively.

   The commands in the `gdb_init` file prepare a directory containing symbols for debugging.

3. Add a `setup` file containing the GDB server configuration commands to the project:

```
setup
```

```
set sysroot <path to project>/kasperskyos_ping/build/einit/sysroot-symbols
set solib-search-path /opt/KasperskyOS-Community-Edition-Qemu-<version>/sysroot-aarch64-kos
set osabi auto
set scheduler-locking off
set schedule-multiple on
set follow-fork-mode parent
set follow-exec-mode same
set detach-on-fork off

monitor exec kl.ExecMgrEntity
catch exec
continue
detach inferiors 1
remove-inferiors 1

monitor exec <process name>
catch exec
continue
b main
continue
```

   Replace the following parameters:

   - `<version>` and `<path to project>` parameters with the SDK version and the path to the root directory of the example, respectively.
   - `<process name>` with the name of the process being debugged. To find out the name of a process, you can run the solution once and check the `name` field in the log. For example, the log will contain a `name: Hello.1` entry for the Hello process. You can also find out the name of the process by using the gdb command `i os processes` after starting the process being debugged.

4. In the root directory of the example, build its image for further debugging:

```
cmake -B build -D CMAKE_TOOLCHAIN_FILE="$KOSCEDIR/toolchain/share/toolchain-aarch64-kos.cmake" -D
CMAKE_BUILD_TYPE=Debug && cmake --build build --target gdbsim
```

   As a result of the build, the following message will be displayed in the terminal:

```
[KDBG] Waiting for GDB connection infinitely
```

5. In a separate terminal, open the program source code file in Neovim.

6. Start vimspector by pressing debugging key (usually **F5**) or by running the `:call vimspector#Launch()` command.

   As a result, the text "`Initializing debug session KOS attach (Hello) (0)..`" appears in the terminal window and debugging starts.

Adapter reference information

Configuration parameters in the `.vimspector.json` file:

`pid#json`

ID (number) of the process being debugged. When starting debugging, Vimspector may prompt you for the number of the process to which it needs to connect.

`program`

Path to a binary file or to a debug symbols file.

`programRunner`

Command to run the program. Both the path to the script containing the command and the command itself are accepted.

`processSpawner`

Process that creates a new process.

`args`

Additional arguments for starting the program (not applicable when debugging in KasperskyOS).

`gdbPath`

Path to the GDB to be used for debugging.

`gdbServer`

Address of the GDB server.

`setupCommands`

Commands for configuring the GDB before starting a debugging session.

`setupServerCommands`

Commands for configuring the GDB server that are applied after connecting to the server.

`useExtendedRemote`

Use extended debug mode. Default is `true`.

`stopOnEntry`

Stops when connecting to a process or entering `main()` when debugging from startup. Default is `true`.

Developing security policies

Formal specifications of KasperskyOS-based solution components

Solution development includes the creation of formal specifications for its components that form a global picture for the Kaspersky Security Module. A *formal specification of a KasperskyOS-based solution component* (hereinafter referred to as the *formal specification of the solution component*) is comprised of a system of IDL, CDL and EDL descriptions (IDL and CDL descriptions are optional) for this component. These descriptions are used to automatically generate transport code of solution components, and source code of the security module and the initializing program. The formal specifications of solution components are also used as source data for the solution security policy description.

> Just like solution components, the KasperskyOS kernel also has a formal specification (for details, see Methods of KasperskyOS core endpoints).

Each solution component corresponds to an EDL description. In terms of a formal specification, a solution component is a container for components that provide endpoints. Multiple instances of one solution component may be used at the same time, which means that multiple processes can be started from the same executable file. Processes that correspond to the same EDL description are processes of the same class. An EDL description defines the process class name and the top-level component parameters, such as the provided endpoints with one or multiple interfaces, the security interface, and embedded components.

Each embedded component corresponds to a CDL description. This description defines the component name, provided endpoints, security interface, and embedded components. Embedded components can simultaneously provide endpoints, support a security interface, and serve as containers for other components. Each embedded component can provide multiple endpoints with one or more interfaces.

Each interface (including the security interface) is defined in an IDL description. This description defines the interface name, signatures of interface methods, and data types for the parameters of interface methods. The data comprising signatures of interface methods and definitions of data types for parameters of interface methods is referred to as a package.

Processes that do not provide endpoints may only act as clients. Processes that provide endpoints are servers, but they can also act as clients at the same time.

> The formal specification of a solution component does not define how this component will be implemented. In other words, the presence of components in a formal specification of a solution component does not mean that these components will be present in the architecture of this solution component.

Names of process classes, components, packages and interfaces

Process classes, components, packages, and interfaces are identified by name. Within one KasperskyOS-based solution, the names of process classes and the names of components form one set of names, while the names of packages form a different set of names. These two sets may overlap. A set of package names includes a set of interface names.

The name of a process class, component, package, or interface is the name of an IDL, CDL, or EDL file to which a prefix can be optionally added. The prefix and name of an IDL, CDL, or EDL file are separated by a dot. The prefix can also consist of several segments, separated by a dot. For example, the `kl.core.DCM` process class name is prefixed with `kl.core`, and the name of the EDL file is `DCM`. Prefixes let you define namespaces to prevent name conflicts in the formal specifications of solution components. Prefix segments can be, for example, the names of functional subsystems or the identifiers of developer teams. However, you should keep in mind that the names of process classes, components, packages, and interfaces in IDL, CDL, or EDL files provided in KasperskyOS Community Edition SDK contain the `kl` prefix or a prefix that begins with the `kl` segment.

> The name of an IDL, CDL or EDL file begins with an uppercase letter and must not contain any underscores `_`.

For source code generators, the name of a process class, component, package, or interface is a link to an IDL, CDL, or EDL file. This link represents the path to an IDL, CDL, or EDL file relative to a directory that is included in a set of directories where source code generators search for IDL, CDL, and EDL files. (This set of directories is defined by the build system.) For example, the `kl.core.DCM` process class name corresponds to the relative path `kl/core/DCM.edl`. The `kl` directory is located in the `sysroot-*-kos/include` directory from KasperskyOS Community Edition SDK, which is included in the set of directories where source code generators search for IDL, CDL, and EDL files.

EDL description

EDL descriptions are placed into separate `*.edl` files and contain declarations in the Entity Definition Language (EDL):

1. **Process class name.** Uses any of the following declarations:

   ```
   task class <process class name>
   // Obsolete and not recommended for use.
   entity <process class name>
   ```

2. [Optional] **List of component instances.** The following declaration is used:

   ```
   components {
       <component instance name> : <component name>
       ...
   }
   ```

   Each component instance is indicated in a separate line. The component instance name must not contain any underscores `_`. The list can contain multiple instances of one component. Each component instance in the list has a unique name.

3. [Optional] **Security interface.** The following declaration is used:

```
security <interface name>
```

4. [Optional] **List of endpoints.** The following declaration is used:

```
endpoints {
    <endpoint name : interface name>
    ...
}
```

   Each endpoint is indicated in a separate line. The endpoint name must not contain any underscores `_`. The list can contain multiple endpoints with the same interface. Each endpoint in the list has a unique name.

The EDL language is case sensitive.

Single-line comments and multi-line comments can be used in an EDL description.

> A security interface and provided endpoints can be defined in an EDL description and in a CDL description. If solution component development is utilizing already prepared constituent parts (such as libraries) that are accompanied by CDL descriptions, it is advisable to refer to them from the EDL description by using the `components` declaration. Otherwise, you can describe all provided endpoints in the EDL description. In addition, you can separately define the security interface in the EDL description and in each CDL description.

Examples of EDL files

Hello.edl

```
// Class of processes that do not contain components.
task class Hello
```

Signald.edl

```
/* Class of processes that contain
 * one instance of one component. */
task class kl.Signald

components {
    signals : kl.Signals
}
```

LIGHTCRAFT.edl

```
/* Class of processes that contain
 * two instances of different components. */
task class kl.drivers.LIGHTCRAFT

components {
    KUSB : kl.drivers.KUSB
    KIDF : kl.drivers.KIDF
}
```

Downloader.edl

```
/* Class of processes that do not contain
 * components and provide one endpoint. */
task class updater.Downloader

endpoints {
    download : updater.Download
}
```

CDL description

CDL descriptions are placed into individual `*.cdl` files and contain declarations in the Component Definition Language (CDL):

1. **The name of the component.** The following declaration is used:

   ```
   component <component name>
   ```

2. [Optional] **Security interface.** The following declaration is used:

   ```
   security <interface name>
   ```

3. [Optional] **List of endpoints.** The following declaration is used:

   ```
   endpoints {
       <endpoint name : interface name>
       ...
   }
   ```

   Each endpoint is indicated in a separate line. The endpoint name must not contain any underscores `_`. The list can contain multiple endpoints with the same interface. Each endpoint in the list has a unique name.

4. [Optional] **List of nested component instances.** The following declaration is used:

   ```
   components {
       <component instance name : component name>
       ...
   }
   ```

   Each component instance is indicated in a separate line. The component instance name must not contain any underscores `_`. The list can contain multiple instances of one component. Each component instance in the list has a unique name.

The CDL language is case sensitive.

Single-line comments and multi-line comments can be used in a CDL description.

> At least one optional declaration is used in a CDL description. If a CDL description does not use at least one optional declaration, this description will correspond to an "empty" component that does not provide endpoints, does not contain embedded components, and does not support a security interface.

Examples of CDL files

KscProductEventsProvider.cdl

```
// Component provides one endpoint.
component kl.KscProductEventsProvider

endpoints {
    eventProvider : kl.IKscProductEventsProvider
}
```

KscConnectorComponent.cdl

```
// Component provides multiple endpoints.
component kl.KscConnectorComponent

endpoints {
    KscConnCommandSender : kl.IKscConnCommandSender
    KscConnController : kl.IKscConnController
    KscConnSettingsHolder : kl.IKscConnSettingsHolder
    KscDataProvider : kl.IKscDataProvider
    ProductDataHolder : kl.IProductDataHolder
    KscDataNotifier : kl.IKscDataNotifier
    KscConnectorStateNotifier : kl.IKscConnectorStateNotifier
}
```

FsVerifier.cdl

```
/* Component does not provide endpoints, supports
 * a security interface, and contains one instance
 * of another component. */
component FsVerifier

security Approve

components {
    verifyComp : Verify
}
```

IDL description

IDL descriptions are placed into separate `*.idl` files and contain declarations in the Interface Definition Language (IDL):

1. **Package name.** The following declaration is used:

   ```
   package <package name>
   ```

2. [Optional] **Packages from which the data types for interface method parameters are imported.** The following declaration is used:

   ```
   import <package name>
   ```

3. [Optional] Definitions of data types for parameters of interface methods.

4. [Optional] **Signatures of interface methods.** The following declaration is used:

```
interface {
    <interface method name>([<parameters>]);
    ...
}
```

Each method signature is indicated in a separate line. The method name must not contain any underscores `_`. Each method in the list has a unique name. The parameters of methods are divided into input parameters (`in`), output parameters (`out`), and parameters for transmitting error information (`error`). The order of parameters in the description is important: first input parameters, then output parameters, then error parameters. Methods of the security interface cannot have output parameters and error parameters.

Input parameters and output parameters are transmitted in IPC requests and IPC responses, respectively. Error parameters are transmitted in IPC responses if the server cannot correctly handle the corresponding IPC requests.

The server can inform a client about IPC request processing errors via error parameters as well as through output parameters of interface methods. If the server sets the error flag in an IPC response when an error occurs, this IPC response will contain the error parameters without any output parameters. Otherwise this IPC response will contain output parameters just like when requests are correctly processed. (The error flag is set in IPC responses by using the `nk_err_reset()` macro defined in the `nk/types.h` header file from the KasperskyOS SDK.)

An IPC response sent with the error flag set and an IPC response with the error flag not set are considered to be different types of events for the Kaspersky Security Module. When describing a solution security policy, this difference lets you conveniently distinguish between the processing of events associated with the correct execution of IPC requests and the processing of events associated with incorrect execution of IPC requests. If the server does not set the error flag in IPC responses, the security module must check the values of output parameters indicating errors to properly process events related to the incorrect execution of IPC requests. (A client can check the state of the error flag in an IPC response even if the corresponding interface method does not contain error parameters. To do so, the client uses the `nk_msg_check_err()` macro defined in the `nk/types.h` header file from the KasperskyOS SDK.)

Signatures of interface methods cannot be imported from other IDL files.

The IDL language is case sensitive.

Single-line comments and multi-line comments can be used in an IDL description.

> At least one optional declaration is used in a IDL description. If an IDL description does not use at least one optional declaration, this description will correspond to an "empty" package that does not assign any interface methods or data types (including from other IDL descriptions).
> Some IDL files from the KasperskyOS SDK do not describe interface methods, but instead only contain definitions of data types. These IDL files are used only as exporters of data types.
> If a package contains a description of interface methods, the interface name matches the package name.

Examples of IDL files

Slog.idl

```
package test.Slog

// Definition of data type for an interface method parameter
typedef sequence<UInt8, 1024> message;

// Interface includes one method.
```

```
interface {
    write(in UInt32 id, in UInt32 reason, in message msg, out SInt32 error);
}
```

Kpm.idl

```
package kl.Kpm

// Import data types for parameters of interface methods
import kl.core.Types

// Definition of data type for parameters of interface methods
typedef string<64> String;

/* Interface includes multiple methods.
 * Some methods do not have any parameters. */
interface {
    Shutdown();
    Reboot();
    PowerButtonPressedWait();
    TerminationSignalWait(in UInt32 entityId, in String entityName);
    EntityTerminated(in UInt32 entityId);
    Terminate(in UInt32 callingEntityId);
}
```

MessageBusSubs.idl

```
package kl.MessageBusSubs

// Import data types for interface method parameters
import kl.MessageBusTypes

/* Interface includes a method that has
 * input and output parameters, and
 * an error parameter.*/
interface {
    Wait(in ClientId id,
         out Message topic,
         out BundleId dataId,
         error ResultCode result);
}
```

WaylandTypes.idl

```
// Package contains only definitions of data types.
package kl.WaylandTypes

typedef UInt32                  ClientId;
typedef bytes<8192>             Buffer;
typedef string<4096>            ConnectionId;
typedef SInt32                  SsizeT;
typedef UInt32                  SizeT;
typedef SInt32                  ShmFd;
typedef SInt32                  ShmId;
typedef bytes<16384000>         ShmBuffer;
```

IDL data types

IDL supports primitive data types as well as composite data types. The set of supported composite types includes unions, structures, arrays, and sequences.

Primitive types

IDL supports the following primitive types:

- `SInt8`, `SInt16`, `SInt32`, `SInt64` – signed integer.
- `UInt8`, `UInt16`, `UInt32`, `UInt64` – unsigned integer.
- `UIntSize` – unsigned integer for representing the object size (in bytes).

  The range of values depends on the hardware platform (equivalent to the `size_t` type in C).
- `UIntPtr` – unsigned integer for representing the object address.

  The range of values depends on the hardware platform (equivalent to the `uintptr_t` type in C).
- `Handle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field.

  A handle can identify any resource.
- `handle<<interface name>>` – value whose binary representation consists of multiple fields, including a callable handle field and a callable handle permissions mask field.

  A callable handle identifies both the IPC channel to the server and the endpoint of this server with the specified interface. The Kaspersky Security Module source code generator verifies that the specified interface is defined in one of the IDL files included in the formal specifications of the solution components. If there is no definition, the generator will terminate with an error.
- `bytes<<maximum size in bytes>>` – byte buffer containing no more than the defined number of bytes.
- `string<<maximum size in bytes>>` – string buffer consisting of a byte buffer whose actual maximum size is a unit larger than the specified size due to the presence of an additional byte for the terminating zero.

  Characters in the string buffer must have UTF-8 encoding. A string buffer can contain a zero-length string without a terminating zero, which indicates the absence of data or an empty string. A non-zero-sized string must be terminated with a terminating zero. (If an encoding other than UTF-8 is used, or if there is no terminating zero in a non-zero-sized string, the Kaspersky Security Module will prevent the transfer of the string buffer.)

Integer literals can be specified in decimal format, hexadecimal format (for example, `0x2f`, `0X2f`, `0x2F`, `0X2F`) or octal format (for example, `00123`, `0o123`).

You can use the reserved word `const` to define the named integer constants by assigning their values using integer literals or integer expressions.

Example definitions of named integer constants:

```
const UInt32 DeviceNameMax = 0o100;
const UInt32 HandleTypeUserLast = 0x0001FFFF;
const UInt32 MaxLogMessageSize = (2 << 3) ** 2;
const UInt32 MaxLogMessageCount = 100;
const UIntSize MaxLen = (MaxLogMessageSize + 4) * MaxLogMessageCount;
```

Named integer constants can be used to avoid problems associated with so-called "magic numbers". For example, if an IDL description defines named integer constants for return codes of an interface method, you can interpret these codes without additional information when describing a policy. Named integer constants and integer expressions can also be applied in definitions of byte buffers, string buffers, and composite types to define the size of data or the number of data elements.

The `bytes<<size in bytes>>`, `string<<size in bytes>>`, and `handle<<interface name>>` constructs are used in definitions of composite types, signatures of interface methods, and when creating type aliases because they define anonymous types (types without a name).

Unions

A union stores different types of data in one memory area. In an IPC message, a union is provided with an additional `tag` field that defines which specific member of the union is used.

The following construct is used to define a union:

```
union <type name> {
    <member type> <member name>;
    ...
}
```

Example of a union definition:

```
union ExitInfo {
    UInt32        code;
    ExceptionInfo exc;
}
```

Structures

The following construct is used to define a structure:

```
struct <type name> {
    <field type> <field name>;
    ...
}
```

Example of a structure definition:

```
struct SessionEvqParams {
    UInt32 count;
    UInt32 align;
    UInt32 size;
}
```

Arrays

The following construct is used to define an array:

```
array<<type of elements>, <number of elements>>
```

This construct is used in definitions of other composite types, signatures of interface methods, and when creating type aliases because it defines an anonymous type.

> The `Handle` and `handle<<interface name>>` types can be used as the type of array elements if this array is not included in another composite data type. However, the total number of handles in an IPC message cannot exceed 255.

Sequences

A sequence is an array containing no more than the specified number of elements of the defined type. The following construct is used to define a sequence:

```
sequence<<type of elements>, <maximum number of elements>>
```

This construct is used in definitions of other composite types, signatures of interface methods, and when creating type aliases because it defines an anonymous type.

> The `Handle` and `handle<<interface name>>` types cannot be used as types of sequence elements.

Variable-size and fixed-size types

The `bytes`, `string` and `sequence` types are variable-size types. In other words, the maximum number of elements is assigned when defining these types, but less elements (or none) may actually be used. Data of the `bytes`, `string` and `sequence` types are stored in the IPC message arena. All other types are fixed-size types. Data of fixed-size types are stored in the constant part of IPC messages.

Types based on composite types

Composite types can be used to define other composite types. The definition of an array or sequence can also be included in the definition of another type.

Example definition of a structure with embedded definitions of an array and sequence:

```
const UInt32 MessageSize = 64;

struct BazInfo {
    array<UInt8, 100> a;
    sequence<sequence<UInt32, MessageSize>, ((2 << 2) + 2 ** 2) * MessageSize> b;
    string<100> c;
    bytes<4096> d;
    UIntSize e;
    handle<embedder.DeviceManager> f;
}
```

The definition of a union or structure cannot be included in the definition of another type. However, a type definition may include already defined unions and structures. This is done by indicating the names of the included types in the type definition.

Example definition of a structure that includes a union and structure:

```
union foo {
    UInt32 value1;
    UInt8 value2;
}

struct bar {
    UIntPtr a;
    UInt8 b;
}

struct BazInfo {
    foo x;
    bar y;
}
```

Creating aliases of types

Type aliases make it more convenient to work with types. For example, type aliases can be used to assign mnemonic names to types that have abstract names. Assigned aliases for anonymous types also let you receive named types.

The following construct is used to create a type alias:

```
typedef <type> <type alias>
```

Example of creating mnemonic aliases:

```
typedef UInt64 ApplicationId;
typedef Handle PortHandle;
```

Example of creating an alias for an array definition:

```
typedef array<UInt8, 4> IP4;
```

Example of creating an alias for a sequence definition:

```
const UInt32 MaxDevices = 8;
struct Device {
    string<32> DeviceName;
    UInt8 DeviceID;
}

typedef sequence<Device, MaxDevices> Devices;
```

Example of creating an alias for a union definition:

```
union foo {
    UInt32 value1;
    UInt8 value2;
}

typedef foo bar;
```

Defining anonymous types in signatures of interface methods

Anonymous types can be defined in signatures of interface methods.

Example of defining a sequence in an interface method signature:

```
const UInt8 DeviceCount = 8;

interface {
    Poll(in UInt32 timeout,
         out sequence<UInt32, DeviceCount / 2> report,
         out UInt32 count,
         out UInt32 rc);
}
```

Integer expressions in IDL

Integer expressions in IDL are composed of named integer constants, integer literals, operators (see the table below), and grouping parentheses.

Example use of integer expressions:

```
const UInt8 itemHeaderLen = 2;
const UInt8 itemBlockLen = 4;
const UInt8 maxItemCount = 0X10;
const UInt64 maxLen = (2 << 3) + (itemHeaderLen + itemBlockLen * 4) * maxItemCount;

interface {
    CopyPage(in array<UInt8, 4 * maxLen> page);
}
```

If an integer overflow occurs when computing an expression, the source code generator using the IDL file will terminate with an error.

Details on operators of integer expressions in IDL

Syntax	Operation	Precedence	Associativity	Special considerations
`-a`	Sign change	1	No	N/A
`~a`	Bitwise negation	1	No	N/A

`a ** b`	Exponentiation	2	No	Special considerations:
				• Due to the lack of associativity, you must use parentheses when specifying multiple consecutive operators to define the order of operations.
				Example:
				`(a ** b) ** c`
				`a ** (b ** c)`
				• The power exponent cannot be negative.

`a * b`	Multiplication	3	Left	N/A
`a / b`	Integer division	3	Left	Special considerations: • The result of the operation is a value obtained by rounding the real quotient down to the next lower integer. For example, *4 / 3 = 1, -4 / 3 = -2*. • The divisor cannot be zero.
`a % b`	Modulo	3	Left	Special considerations: • The sign of the operation result matches the sign of the divisor. For example, *-5 % 2 = 1, 5 % -2 = -1*. • The divisor cannot be zero.
`a + b`	Addition	4	Left	N/A
`a - b`	Subtraction	4	Left	N/A

`a << b`	Bit shift left	2*	No	Special considerations:
				• The precedence is lower than with unary operations but incomparable to the precedences of binary arithmetic operations. Therefore, in expressions with binary arithmetic operations, you must use parentheses to define the order of operations.
				Example:
				`a << (b + c)`
				`(a << b) ** c`
				• Due to the lack of associativity, you must use parentheses when specifying multiple consecutive operators to define the order of operations.
				Example:
				`(a << b) << c`
				`a << (b << c)`
				• The shift value must be within the interval *[0; 63]*.

Syntax	Operation	Precedence	Associativity	Special considerations

`a >> b`	Bit shift right	2*	No	Special considerations:
				• The precedence is lower than with unary operations but incomparable to the precedences of binary arithmetic operations. Therefore, in expressions with binary arithmetic operations, you must use parentheses to define the order of operations.
				Example:
				`a >> (b * c)`
				`(a >> b) / c`
				• Due to the lack of associativity, you must use parentheses when specifying multiple consecutive operators to define the order of operations.
				Example:
				`(a >> b) >> c`
				`a >> (b >> c)`
				• The shift value must be within the interval *[0; 63]*.

Describing a security policy for a KasperskyOS-based solution

A *KasperskyOS-based solution security policy description* (hereinafter also referred to as a *solution security policy description* or *policy description*) provides a set of interrelated text files with the `psl` extension that contain declarations in the PSL language (Policy Specification Language). Some files reference other files through an inclusion declaration, which results in a hierarchy of files with one top-level file. The top-level file is specific to the solution. Files of lower and intermediate levels contain parts of the solution security policy description that may be specific to the solution or may be re-used in other solutions.

Some of the files of the lower and intermediate levels are provided in the KasperskyOS SDK. These files contain definitions of the basic data types and formal descriptions of KasperskyOS security models. *KasperskyOS security models* (hereinafter referred to as *security models*) serve as the framework for implementing security policies for KasperskyOS-based solutions. Files containing formal descriptions of security models reference a file containing definitions of the basic data types that are used in the descriptions of models.

The other files of lower and intermediate levels are created by the policy description developer if any parts of the policy description need to be re-used in other solutions. A policy description developer can also put parts of the policy description into separate files for convenient editing.

The top-level file references files containing definitions of basic data types and descriptions of security models that are applied in the part of the solution security policy that is described in this file. The top-level file also references all files of the lower and intermediate levels that were created by the policy description developer.

The KasperskyOS boot image includes a description of the security policy for system programs included in the boot image. This policy is automatically added to the solution security policy during the build. For more details, refer to Architecture and startup of a KasperskyOS-based solution.

> The top-level file is normally named `security.psl`, but it can have any other name in the `*.psl` format.

General information about a KasperskyOS-based solution security policy description

In simplified terms, a KasperskyOS-based solution security policy description consists of bindings that associate descriptions of security events with calls of methods provided by security model objects. A *security model object* is an instance of a class whose definition is a formal description of a security model (in a PSL file). Formal descriptions of security models contain signatures of *methods of security models* that determine the permissibility of interactions between different processes and between processes and the KasperskyOS kernel. These methods are divided into two types:

- *Security model rules* are methods of security models that return a "granted" or "denied" result. Security model rules can change security contexts (for information about a security context, see Resource Access Control).
- *Security model expressions* are methods of security models that return values that can be used as input data for other methods of security models.

A security model object provides methods that are specific to one security model and stores the parameters used by these methods (for example, the initial state of a finite-state machine or the size of a container for specific data). The same object can be used to work with multiple resources. (In other words, you do not need to create a separate object for each resource.) However, the security contexts of these resources will be independent of each other. Likewise, multiple objects of one or more different security models can be used to work with the same resource. In this case, different objects will use the security context of the same resource without any reciprocal influence.

Security events serve as signals indicating the initiation of interaction between different processes and between processes and the KasperskyOS kernel. Security events include the following events:

- Clients send IPC requests.
- Servers or the kernel send IPC responses.
- The kernel or processes initialize the startup of processes.
- The kernel starts.
- Processes query the Kaspersky Security Module via the security interface.

Security events are processed by the security module.

Security models

The KasperskyOS SDK provides PSL files that describe the following security models:

- Base – methods that implement basic logic.
- Pred – methods that implement comparison operations.
- Bool – methods that implement logical operations.

- Math – methods that implement integer arithmetic operations.
- Struct – methods that provide access to structural data elements (for example, access to parameters of interface methods transmitted in IPC messages).
- Regex – methods for text data validation based on regular expressions.
- HashSet – methods for working with one-dimensional tables associated with resources.
- StaticMap – methods for working with two-dimensional "key–value" tables associated with resources.
- Flow – methods for working with finite-state machines associated with resources.
- Mic – methods for implementing *Mandatory Integrity Control* (MIC).

Security event handling by the Kaspersky Security Module

The Kaspersky Security Module calls all methods (rules and expressions) of security models related to an occurring security event. If all rules returned the "granted" result, the security module returns the "granted" decision. If even one rule returned the "denied" result, the security module returns the "denied" decision.

If even one method related to an occurring security event cannot be correctly performed, the security module returns the "denied" decision.

If no rule is related to an occurring security event, the security module returns the "denied" decision. In other words, all interactions between solution components and between those components and the KasperskyOS kernel are denied by default (Default Deny principle) unless those interactions are explicitly allowed by the solution security policy.

PSL language syntax

Basic rules

1. Declarations can be listed in any sequence in a file.
2. One declaration can be written to one or multiple lines.
3. The PSL language is case sensitive.
4. Single-line comments and multi-line comments are supported:

```
/* This is a comment
 * And this, too */
// Another comment
```

Types of declarations

The PSL language has declarations for the following purposes:

- Setting the global parameters of a solution security policy
- Including PSL files in a solution security policy description
- Including EDL files in a solution security policy description
- Create security model objects
- Bind methods of security models to security events
- Creating security audit profiles
- Creating solution security policy tests

Setting the global parameters of a KasperskyOS-based solution security policy

Global parameters include the following parameters of a solution security policy:

- *Execute interface* used by the KasperskyOS kernel when querying the Kaspersky Security Module to notify it about kernel startup or about initiating the startup of a process by the kernel or by other processes. To define this interface, use the following declaration:

  ```
  execute: kl.core.Execute
  ```

  KasperskyOS currently supports only one execute interface (`Execute`) defined in the file named `kl/core/Execute.idl`. (This interface consists of one `main` method, which has no parameters and does not perform any actions. The `main` method is reserved for potential future use.)

- [Optional] Global security audit profile and initial security audit runtime-level. (For more details about profiles and the security audit runtime-level, see Creating security audit profiles.) To define these parameters, use the following declaration:

  ```
  audit default = <security audit profile name> <security audit runtime-level>
  ```

  Example:

  ```
  audit default = global 0
  ```

  The default global profile is the `empty` security audit profile described in the file named `toolchain/include/nk/base.psl` from the KasperskyOS SDK, and the default security audit runtime-level is 0. When the `empty` security audit profile is applied, a security audit is not conducted.

Including PSL files in a KasperskyOS-based solution security policy description

To include a PSL file in a policy description, use the following declaration:

```
use <link to PSL file>._
```

The link to the PSL file is the path to the PSL file (without the extension and the dot in front of it) relative to the directory where the `*.psl[.in]` file is located, whose path is specified in the `SECURITY_PSL_FILE` parameter of the CMake command `build_vendor_romfs ()`. A dot is used as a separator in a path description. A declaration is ended by the `._` character sequence.

Example:

```
use policy_parts.flow_part._
```

This declaration includes the `<path>/policy_parts/flow_part.psl` file in the policy description if the `SECURITY_PSL_FILE` parameter of the CMake command `build_vendor_romfs ()` is set to `<path>/*.psl[.in]`.

Including a PSL file containing a formal description of a security model

To use the methods of a required security model, the policy description must include a PSL file containing a formal description of this model. PSL files containing formal descriptions of security models are located in the KasperskyOS SDK at the following path:

`toolchain/include/nk`

Example:

```
/* Include the base.psl file containing a formal description of the
 * Base security model */
use nk.base._

/* Include the flow.psl file containing a formal description of the
 *  Flow security model */
use nk.flow._
```

Including EDL files in a KasperskyOS-based solution security policy description

To include an EDL file for the KasperskyOS kernel in a policy description, use the following declaration:

```
use EDL kl.core.Core
```

To include an EDL file for a program (such as a driver or application) into a policy description, use the following declaration:

```
use EDL <process class name>
```

Example:

```
/* Include the UART.edl file */
use EDL kl.drivers.UART

/* Include the Einit.edl file */
use EDL Einit

/* Include the Server.edl file */
use EDL ping.Server
```

The Kaspersky Security Module source code generator finds IDL and CDL files through EDL files because EDL files contain the names of components and interfaces that are links to IDL and CDL files. CDL files also contain the names of components and interfaces for searching IDL and CDL files.

Creating security model objects

To call the methods of a required security model, create an object for this security model.

To create a security model object, use the following declaration:

```
policy object <security model object name> : <security model name> {
    [<security model object parameters>]
}
```

The security model object name must begin with a lowercase letter. The parameters of a security model object are specific to the security model. A description of parameters and examples of creating objects of various security models are provided in the Kaspersky OS security models section.

Binding methods of security models to security events

To create an attachment between methods of security models and a security event, use the following declaration:

```
<security event type> [<security event selectors>] {
    [<security audit profile>]
    <called security model methods>
}
```

Security event type

To define the security event type, use the following specifiers:

- `request` – sending IPC requests.
- `response` – sending IPC responses.
- `error` – sending IPC responses containing information about errors.
- `security` – processes querying the Kaspersky Security Module via the security interface.
- `execute` – initializing the startups of processes or the startup of the KasperskyOS kernel.

> When processes interact with the security module, they use a mechanism that is different from IPC. However, when describing a policy, queries sent by processes to the security module can be viewed as the transfer of IPC messages because processes actually transmit messages to the security module (the recipient is not indicated in these messages).
> The IPC mechanism is not used to start processes. However, when the startup of a process is initiated, the kernel queries the security module and provides information about the initiator of the startup and the started process. For this reason, the policy description developer can consider the startup of a process to be analogous to sending an IPC message from the startup initiator to the started process. When the kernel is started, this is analogous to the kernel sending an IPC message to itself.

Security event selectors

Security event selectors let you clarify the description of the defined type of security event. You can use the following selectors:

- `src=<kernel/process class name>` – processes of the defined class or the KasperskyOS kernel are the sources of IPC messages.
- `dst=<kernel/process class name>` – processes of the defined class or the kernel are the recipients of IPC messages.
- `interface=<interface name>` – describes the following security events:
  - Clients attempt to use the endpoints of servers or the kernel with the defined interface.
  - Processes query the Kaspersky Security Module via the defined security interface.
  - The kernel or servers send clients the results from using the endpoints with the defined interface.
- `component=<component name>` – describes the following security events:
  - Clients attempt to use the core or server endpoints provided by the defined component.
  - The kernel or servers send clients the results from using the endpoints provided by the defined component.

- `endpoint=<qualified endpoint name>` – describes the following security events:
  - Clients attempt to use the defined core or server endpoints.
  - The kernel or servers send clients the results from using the defined endpoint.
- `method=<method name>` – describes the following security events:
  - Clients attempt to query servers or the kernel by calling the defined method of the endpoint.
  - Processes query the security module by calling the defined method of the security interface.
  - The kernel or servers send clients the results from calling the defined method of the endpoint.
  - The kernel notifies the security module about its startup by calling the defined method of the execute interface.
  - The kernel initiates the startup of processes by calling the defined method of the execute interface.
  - Processes initiate the startup of other processes, which results in the kernel calling the defined method of the execute interface.

Process classes, components, instances of components, interfaces, endpoints, and methods must be named the same as they are in the IDL, CDL, and EDL descriptions. The kernel must be named `kl.core.Core`.

The qualified name of an endpoint has the format `<qualified name of component instance>.<endpoint name>`. The qualified name of a component instance has the format `<path to component instance>.<component instance name>`. The path to a component instance is a sequence of component instance names separated by dots. Among these component instances, each subsequent component instance is embedded into the previous one, and the last one contains the component instance with the defined name.

For `security` events, you must specify the qualified name of the security interface method if you need to use the security interface defined in a CDL description. (If you need to use a security interface defined in an EDL description, it is not necessary to specify the qualified name of the method.) The qualified name of a security interface method is a construct in the format `<qualified name of component instance>.<security interface method name>`.

If selectors are not specified, the participants of a security event may be any process and the kernel (except `security` events in which the kernel cannot participate).

You can use combinations of selectors. Selectors can be separated by commas.

There are restrictions on the use of selectors. The `interface`, `component`, and `endpoint` selectors cannot be used for security events of the `execute` type. The `dst`, `component`, and `endpoint` selectors cannot be used for security events of the `security` type.

There are also restrictions on combinations of selectors. For security events of the `request`, `response` and `error` types, the `method` selector can only be used together with one of the `endpoint`, `interface`, or `component` selectors or a combination of them. (The `method`, `endpoint`, `interface` and `component` selectors must be coordinated. In other words, the method, endpoint, interface, and component must be interconnected.) For security events of the `request` type, the `endpoint` selector can be used only together with the `dst` selector. For security events of the `response` and `error` types, the `endpoint` selector can be used only together with the `src` selector.

> The type and selectors of a security event form the security event description. It is recommended to describe security events with maximum precision to allow only the required interactions between different processes and between processes and the kernel. If IPC messages of the same type are always verified when processing the defined event, the description of this event is maximally precise.

To ensure that IPC messages of the same type correspond to a security event description, one of the following conditions must be fulfilled for this description:

- For events of the `request`, `response` and `error` type, the "interface method-endpoint-server class or kernel" chain is unequivocally defined. For example, the security event description `request dst=Server endpoint=net.Net method=Send` corresponds to IPC messages of the same type, and the security event description `request dst=Server` corresponds to any IPC message sent to the `Server`.
- For `security` events, the security interface method is specified.
- The execute-interface method is indicated for `execute` events.

  There is currently support for only one fictitious method of the `main` execute-interface. This method is used by default, so it does not have to be defined through the `method` selector. This way, any description of an `execute` security event corresponds to IPC messages of the same type.

Security audit profile

To define a security audit profile, use the following construct:

```
audit <security audit profile name>
```

If a security audit profile is not defined, the global security audit profile is used.

Called security model methods

To call a security model method, use the following construct:

```
[<security model object name>.]<security model method name> <parameter>
```

Data of PSL-supported types can be used as the parameter. However, the following special considerations should be taken into account:

- If a security model method does not actually have a parameter, this method formally has a Unit-type parameter designated as `()`.
- If a security model method parameter is a dictionary `{<name of field 1> : <value of field 1>[, <name of field 2> : <value of field 2>]…}`, this parameter does not need to be enclosed in parentheses.
- If the security model method parameter is not a dictionary and does not have the Unit type, this parameter must be enclosed in parentheses.

You can call one or more methods by using the same or different security model objects. Security model rules can use the parameter to receive values returned by expressions of security models.

> When a security event is processed by the Kaspersky Security Module, expressions are called before rules. Therefore, expressions do not receive the changes made by rules. For example, if a declaration of attachment between StaticMap security model methods and security events first specifies the `set` rule and then specifies the `get_uncommited` expression for the same resource, the `get_uncommited` expression will return the key value that was defined before the current security event was processed instead of the key value that is defined by the `set` rule when processing the current security event. The key value defined by the `set` rule when processing the current security event can be returned by the `get_uncommited` expression only when processing subsequent security events if the security module returns the "allowed" decision as a result of processing the current security event. If the security module returns a "denied" decision as a result of processing the current security event, all changes made by rules and expressions invoked during processing of the current security event will be discarded.

A security model method (rule or expression) can use the parameter to receive the parameters of interface methods. (For details about obtaining access to parameters of interface methods, see Struct security model.) A security model method can also use the parameter to receive the SID values of processes and the KasperskyOS kernel that are defined by the reserved words `src_sid` and `dst_sid`. The first reserved word refers to the SID of the process (or kernel) that is the source of the IPC message. The second reserved word refers to the SID of the process (or kernel) that is the recipient of the IPC message (`dst_sid` cannot be used for queries to the Kaspersky Security Module).

> You do not have to indicate the security model object name to call certain security model methods. Also, some of the security model methods must be called using operators instead of the call construct. For details about the methods of security models, see KasperskyOS security models.

Embedded constructs for binding methods of security models to security events

In one declaration, you can bind methods of security models to different security events of the same type. To do so, use the match sections that consist of the following types of constructs:

```
match <security event selectors> {
    [<security audit profile>]
```

```
        <called security model methods>
    }
```

Match sections can be embedded into another match section. A match section simultaneously uses its own security event selectors and the security event selectors at the level of the declaration and all match sections in which this match section is "wrapped". By default, a match section applies the security audit profile of its own container (match section of the preceding level or the declaration level), but you can define a separate security audit profile for the match section.

In one declaration, you can define different variants for processing a security event depending on the conditions in which this event occurred (for example, depending on the state of the finite-state machine associated with the resource). To do so, use the conditional sections that are elements of the following construct:

```
choice <call of the security model expression that verifies fulfillment of conditions> {
    "<condition 1>" : [{] // Conditional section 1
        <called security model methods>
    [}]
    "<condition 2>" : ... // Conditional section 2
...
    _             : ... // Conditional section, if no condition is fulfilled.
}
```

The `choice` construct can be used within a match section. A conditional section uses the security event selectors and security audit profile of its own container.

If multiple conditions described in the `choice` construct are simultaneously fulfilled when a security event is processed, only the one conditional section corresponding to the first true condition on the list is triggered.

> You can verify the fulfillment of conditions in the `choice` construct only by using the expressions that are specially intended for this purpose. Some security models contain these expressions (for more details, see KaspersкyOS security models).
> Only text and integer literals, logical values and the `_` character designating an always true condition can be used as conditions.

Examples of binding security model methods to security events

See "Examples of binding security model methods to security events", "Example descriptions of basic security policies for KaspersкyOS-based solutions", and "KaspersкyOS security models".

Creating security audit profiles

During a *security audit* (hereinafter referred to as an *audit*), the Kaspersky Security Module sends information about its decisions (security audit data) to the logging system, which outputs this data (for example, via the UART) or saves it to a file. *Security audit data* (hereinafter referred to as *audit data*) refers to information about decisions made by the Kaspersky Security Module, which includes the actual decisions ("granted" or "denied"), descriptions of security events, results from calling methods of security models, and data on incorrect IPC messages. Data on calls of security model expressions is included in the audit data just like data on calls of security model rules.

To perform a security audit, you need to associate security model objects with security audit profile(s). A *security audit profile* (hereinafter also referred to as an *audit profile*) combines *security audit configurations* (hereinafter also referred to as *audit configurations*), each of which defines the security model objects covered by the audit, and specifies the conditions for conducting the audit. You can define a global audit profile (for more details, see Setting the global parameters of a KaspershyOS-based solution security policy) and/or assign an audit profile(s) at the level of binding security model methods to security events, and/or assign an audit profile(s) at the level of match sections (for more details, see Binding methods of security models to security events).

> Regardless of whether or not audit profiles are being used, audit data contains information about "denied" decisions that were made by the Kaspersky Security Module when IPC messages were invalid and when handling security events that are not associated with any security model rule.

To create a security audit profile, use the following declaration:

```
audit profile <security audit profile name> =
  { <security audit runtime-level 1> :
      // Security audit configuration 1
      { <security model object name 1> :
          { kss: <security audit conditions linked to the results
                  from calls of security model methods>
           [, <security audit conditions specific to the security model>]
          }
     [, <security model object name 2> :
          {
          ...
          }]
      ...
      }
 [, <security audit runtime-level 2> :
      // Security audit configuration 2
      {
      ...
      }]
  ...
  }
```

Security audit runtime-level

The *security audit runtime-level* (hereinafter referred to as the *audit runtime-level*) is a global parameter of a solution security policy and consists of an unsigned integer that defines the active security audit configuration. (The word "runtime-level" here refers to the configuration variant and does not necessarily involve a hierarchy.) The audit runtime-level can be changed during operation of the Kaspersky Security Module. To do so, use a specialized method of the `Base` security model that is called when processes query the security module through the security interface (for more details, see Base security model). The initial audit runtime-level is assigned together with the global audit profile (for more details, see Setting the global parameters of a KaspershyOS-based solution security policy). An `empty` audit profile can be explicitly assigned as the global audit profile.

You can define multiple audit configurations in an audit profile. In different configurations, different security model objects can be covered by the audit and different conditions for conducting the audit can be applied. Audit configurations in a profile correspond to different audit runtime-levels. If a profile does not have an audit configuration corresponding to the current audit runtime-level, the security module will activate the configuration that corresponds to the next-lowest audit runtime-level. If a profile does not have an audit configuration for an audit runtime-level equal to or less than the current level, the security module will not use this profile (in other words, an audit will not be performed for this profile).

The capability to change the audit runtime-level lets you regulate the level of detail of an audit, for example. The higher the audit runtime-level, the higher the level of detail. In other words, a higher audit runtime-level activates audit configurations in which more security model objects are covered by the audit and/or less restrictions are applied in the audit conditions. In addition, you can change the audit runtime-level to switch the audit from one set of logically connected security model objects to another set. For example, a low audit runtime-level activates audit configurations in which the audit covers security model objects related to drivers, a medium audit runtime-level activates audit configurations in which the audit covers security model objects related to the network subsystem, and a high audit runtime-level activates audit configurations in which the audit covers security model objects related to applications.

Name of the security model object

The security model object name is indicated so that the methods provided by this object can be covered by the audit. These methods will be covered by the audit whenever they are called, provided that the conditions for conducting the audit are observed.

> Information about the decisions of the Kaspersky Security Module contained in audit data includes the overall decision of the security module as well as the results from calling individual methods of security modules covered by the audit. To ensure that information about a security module decision is included in audit data, at least one method called during security event handling must be covered by the audit.
> The names of security model objects and the names of methods provided by these objects are included in the audit data.

Security audit conditions

Security audit conditions must be defined separately for each object of a security model.

To define the audit conditions related to the results from calling security model methods, use the following constructs:

- `["granted"]` – the audit is performed if the rules return the "granted" result; the expressions are correctly executed.
- `["denied"]` – the audit is performed if the rules return the "denied" result; the expressions are incorrectly executed.
- `["granted", "denied"]` – the audit is performed regardless of the result returned by rules, and regardless of whether or not rules are correctly fulfilled.
- `[]` – the audit is not performed.

Audit conditions specific to security models are defined by constructs specific to these models (for more details, see KasperskyOS Security models). These conditions apply to rules and expressions. For example, one of these conditions can be the state of a finite-state machine.

Security audit profile for a security audit route

The Kaspersky Security Module forwards audit data to the kernel. The kernel outputs this data (for example, via the UART) or forwards it to the system server program LogRR. In the second case, the security audit route includes a set of processes connected by IPC channels. For example, the security audit route may be arranged as follows: "kernel – the LogRR server program – LogRR output channel for UART – UART driver". It is not recommended to audit the methods of security models that are associated with data transmission through the security audit route. This may lead to an avalanche of audit data because any data transmission will give rise to new data.

To "suppress" an audit that was defined by a profile with a wider scope (for example, by a global profile or a profile at the level of binding security model methods to a security event), assign an `empty` audit profile at the level of binding security model methods to security events or at the level of the match section.

Examples of security audit profiles

See "Examples of security audit profiles".

PSL data types

The data types supported in the PSL language are presented in the table below.

PSL data types

Designations of types	Description of types
`UInt8`, `UInt16`, `UInt32`, `UInt64`	Unsigned integer
`SInt8`, `SInt16`, `SInt32`, `SInt64`	Signed integer
`Boolean`	Boolean type The Boolean type includes two values: `true` and `false`.
`Text`	Text type
`()`	`Unit` type The `Unit` type includes one immutable value. It is used as a stub value in cases when PSL language syntax requires certain data formulation but this data is not actually required. For example, the Unit type can be used to declare a method that does not have any parameters (similar to how the `void` type is used in C/C++).

`"[<text>]"`	Text literal
	A text literal includes one immutable text value.
	Example definitions of text literals:
	`""`
	`"granted"`

`<integer>`	Integer literal
	An integer literal includes one immutable integer value.
	Example definitions of integer literals:
	`12`
	`-5`
	`0xFFFF`

Designations of types	Description of types

`<type 1> | <type 2> [| <type 3>]…`	Variant type
	A variant type combines two or more types and may perform the role of either of them.
	Examples of definitions of variant types:
	`Boolean | ()`
	`UInt8 | UInt16 | UInt32 | UInt64`
	`"granted" | "denied"`

`{ [<name of field 1> : <type of field 1>]`
`[, <name of field 2> : <type of field 2>]`
`…`
`}`
	Dictionary
	A dictionary consists of one or more types of fields. A dictionary can be empty.
	Examples of dictionary definitions:
	`{}`
	`{ handle : Handle`
	`, rights : UInt32`
	`}`

`[[<type 1>][, <type 2>]…]`	Tuple
	A tuple consists of fields of one or more types in the order in which the types are listed. A tuple can be empty.
	Examples of tuple definitions:
	`[]`
	`["granted"]`
	`[Boolean, Boolean]`

`Set<<type of elements>>`	Set
	A set includes zero or more unique elements of the same type.
	Examples of set definitions:
	`Set<"granted" | "denied">`
	`Set<Text>`

`List<<type of elements>>`	List
	A list includes zero or more elements of the same type.
	Examples of list definitions:
	`List<Boolean>`
	`List<Text | ()>`

`Map<<key type>, <value type>>`	Associative array An associative array includes zero or more entries of the "key-value" type with unique keys. Example of defining an associative array: `Map<UInt32, UInt32>`
`Array<<type of elements>, <number of elements>>`	Array An array includes a defined number of elements of the same type. Example of defining an array: `Array<UInt8, 42>`
`Sequence<<type of elements>, <number of elements>>`	Sequence A sequence includes from zero to the defined number of elements of the same type. Example of defining a sequence: `Sequence<SInt64, 58>`

Aliases of certain PSL types

The `nk/base.psl` file from the KasperskyOS SDK defines the data types that are used as the types of parameters (or structural elements of parameters) and returned values for methods of various security models. Aliases and definitions of these types are presented in the table below.

Aliases and definitions of certain data types in PSL

Type alias	Type definition
`Unsigned`	Unsigned integer `UInt8 | UInt16 | UInt32 | UInt64`
`Signed`	Signed integer `SInt8 | SInt16 | SInt32 | SInt64`
`Number`	Integer `Unsigned | Signed`
`ScalarLiteral`	Scalar literal `() | Boolean | Number`
`Literal`	Literal `ScalarLiteral | Text`
`Sid`	Type of security ID (SID) `UInt32`
`Handle`	Type of security ID (SID) `Sid`

`HandleDesc`	Dictionary containing fields for the SID and handle permissions mask
	`{ handle : Handle`
	`, rights : UInt32`
	`}`

`Cases`	Type of data received by expressions of security models called in the `choice` construct for verifying fulfillment of conditions `List<Text | ()>`
`KSSAudit`	Type of data defining the conditions for conducting the security audit `Set<"granted" | "denied">`

Mapping IDL types to PSL types

Data types of the IDL language are used to describe the parameters of interface methods. The input data for security model methods have types from the PSL language. The set of data types in the IDL language differs from the set of data types in the PSL language. Parameters of interface methods transmitted in IPC messages can be used as input data for methods of security models, so the policy description developer needs to understand how IDL types are mapped to PSL types.

Integer types of IDL are mapped to integer types of PSL and to variant types of PSL that combine these integer types (including with other types). For example, signed integer types of IDL are mapped to the `Signed` type in PSL, and integer types of IDL are mapped to the `ScalarLiteral` type in PSL.

The `Handle` type in IDL is mapped to the `HandleDesc` type in PSL.

Unions and structures of IDL are mapped to PSL dictionaries.

Arrays and sequences of IDL are mapped to arrays and sequences of PSL, respectively.

String buffers in IDL are mapped to the text type in PSL.

Byte buffers in IDL are not currently mapped to PSL types, so the data contained in byte buffers cannot be used as inputs for security model methods.

Examples of binding security model methods to security events

> Before analyzing examples, you need to become familiar with the Base security model.

Processing the initiation of process startups

```
/* The KasperskyOS kernel and any process
 * in the solution is allowed to start any
 * process. */
execute { grant () }

/* The kernel is allowed to start a process
 * of the Einit class. */
execute src=kl.core.Core, dst=Einit { grant () }

/* An Einit-class process is allowed
 * to start any process in the solution. */
execute src=Einit { grant () }
```

Handling the startup of the KasperskyOS kernel

```
/* The KasperskyOS kernel is allowed to start.
 * (This binding is necessary so that the security
 * module can be notified of the kernel SID. The kernel starts irrespective
 * of whether this is allowed by the solution security policy
 * or denied. If the solution security policy denies the
 * startup of the kernel, after startup the kernel will terminate its
 * execution.) */
execute src=kl.core.Core, dst=kl.core.Core { grant () }
```

Handling IPC request forwarding

```
/* Any client in the solution is allowed to query
 * any server and the KasperskyOS kernel. */
request { grant () }

/* A client of the Client class is allowed to query
 * any server in the solution and the kernel. */
request src=Client { grant () }

/* Any client in the solution is allowed to query
 * a server of the Server class. */
request dst=Server { grant () }

/* A client of the Client class is not allowed to
 * query a server of the Server class. */
request src=Client dst=Server { deny () }

/* A client of the Client class is allowed to
 * query a server of the Server class
 * by calling the Ping method of the net.Net endpoint. */
```

```
request src=Client dst=Server endpoint=net.Net method=Ping {
    grant ()
}

/* Any client in the solution is allowed to query
 * a server of the Server class by calling the Send method
 * of the endpoint with the MessExch interface. */
request dst=Server interface=MessExch method=Send {
    grant ()
}
```

Handling IPC response forwarding

```
/* A server of the Server class is allowed to respond to
 * queries of a Client-class client that
 * calls the Ping method of the net.Net endpoint. */
response src=Server, dst=Client, endpoint=net.Net, method=Ping {
    grant ()
}

/* The server containing the kl.drivers.KIDF component
 * that provide endpoints with the monitor interface is allowed to
 * respond to queries of a DriverManager-class client
 * that uses these endpoints. */
response dst=DriverManager component=kl.drivers.KIDF interface=monitor {
    grant ()
}
```

Handling the transmission of IPC responses containing error information

```
/* A server of the Server class is not allowed to notify a client
 * of the Client class regarding errors that occur
 * when the client queries the server by calling the
 * Ping method of the net.Net endpoint. */
error src=Server, dst=Client, endpoint=net.Net, method=Ping {
    deny ()
}
```

Handling queries sent by processes to the Kaspersky Security Module

```
/* A process of the Sdcard class will receive the
 * "granted" decision from the Kaspersky Security Module
 * by calling the Register method of the security interface.
 * (Using the security interface defined
 * in the EDL description.) */
security src=Sdcard, method=Register {
    grant ()
}

/* A process of the Sdcard class will receive the "denied" decision
 * from the security module when calling the Comp.Register method
 * of the security interface. (Using the security interface
 * defined in the CDL description.) */
security src=Sdcard, method=Comp.Register {
    deny ()
}
```

Using match sections

```
/* A client of the Client class is allowed to query
 * a server of the Server class by calling the Send
 * and Receive methods of the net endpoint. */
request src=Client, dst=Server, endpoint=net {
    match method=Send { grant () }
    match method=Receive { grant () }
}

/* A client of the Client class is allowed to query
 * a server of the Server class by calling the Send
 * and Receive methods of the sn.Net endpoint and the Write and
 * Read methods of the sn.Storage endpoint. */
request src=Client, dst=Server {
    match endpoint=sn.Net {
        match method=Send { grant () }
        match method=Receive { grant () }
    }
    match endpoint=sn.Storage {
        match method=Write { grant () }
        match method=Read { grant () }
    }
}
```

Setting audit profiles

```
/* Set the default global audit profile
 * and initial audit runtime-level of 0 */
audit default = global 0
request src=Client, dst=Server {
    /* Set the parent audit profile at the level of
     * binding methods of security models to
     * security events */
    audit parent
    match endpoint=net.Net, method=Send {
        /* Set a child audit profile at the
         * match section level */
        audit child
        grant ()
    }
    /* This match section applies a
     * parent audit profile. */
    match endpoint=net.Net, method=Receive {
        grant ()
    }
}
/* This binding of the security model method
 * to the security event utilizes the
 * global audit profile. */
response src=Client, dst=Server {
     grant ()
}
```

Example descriptions of basic security policies for KasperskyOS-based solutions

> Before analyzing examples, you need to become familiar with the Struct, Base and Flow security models.

Example 1

The solution security policy in this example allows any interaction between different processes of the `Client`, `Server` and `Einit` classes, and between these processes and the KasperskyOS kernel. The "granted" decision will always be received when these processes query the Kaspersky Security Module.

This policy can be used only as a stub during the early stages of development of a KasperskyOS-based solution so that the Kaspersky Security Module does not interfere with interactions. It would be unacceptable to apply such a policy in a real-world KasperskyOS-based solution.

security.psl

```
execute: kl.core.Execute

use nk.base._
use EDL Einit
use EDL Client
use EDL Server
use EDL kl.core.Core

execute { grant () }

request { grant () }

response { grant () }

error { grant () }

security { grant () }
```

Example 2

The solution security policy in this example imposes limitations on queries sent from clients of the `FsClient` class to servers of the `FsDriver` class. When a client opens a resource controlled by a server of the `FsDriver` class, a finite-state machine in the `unverified` state is associated with this resource. A client of the `FsClient` class is allowed to read data from a resource controlled by a server of the `FsDriver` class only if the finite-state machine associated with this resource is in the `verified` state. To switch a resource-associated finite-state machine from the `unverified` state to the `verified` state, a process of the `FsVerifier` class needs to query the Kaspersky Security Module.

In a real-world KasperskyOS-based solution, this policy cannot be applied because it allows an excessive variety of interactions between different processes and between processes and the KasperskyOS kernel.

security.psl

```
execute: kl.core.Execute

use nk.base._
use nk.flow._
use nk.basic._

policy object file_state : Flow {
    type States = "unverified" | "verified"
    config = {
        states      : ["unverified" , "verified"],
        initial     : "unverified",
        transitions : {
```

```
            "unverified" : ["verified"],
            "verified" : []
        }
    }
}


execute { grant () }

request { grant () }

response { grant () }

use EDL kl.core.Core
use EDL Einit
use EDL FsClient
use EDL FsDriver
use EDL FsVerifier

response src=FsDriver, endpoint=operationsComp.operationsImpl, method=Open {
    file_state.init {sid: message.handle.handle}
}

request src=FsClient, dst=FsDriver, endpoint=operationsComp.operationsImpl, method=Read {
    file_state.allow {sid: message.handle.handle, states: ["verified"]}
}

security src=FsVerifier, method=Approve {
    file_state.enter {sid: message.handle.handle, state: "verified"}
}
```

Examples of security audit profiles

> Before analyzing examples, you need to become familiar with the Base, Regex and Flow security models.

Example 1

```
// Describing a trace security audit profile
// base – Base security model object
// session – Flow security model object
audit profile trace =
/* If the audit runtime-level is equal to 0, the audit covers
 * base object rules when these rules return
 * the "denied" result. */
  { 0 :
      { base :
          { kss : ["denied"]
          }
      }
/* If the audit runtime-level is equal to 1, the audit covers methods
* of the session object in the following cases:
* 1.Rules of the session object return any result, and
 * the finite-state machine is in a state other than closed.
* 2.A query expression of the session object is executed, and the
* finite-state machine is in a state other than closed. */
  , 1 :
      { session :
          { kss : ["granted", "denied"]
          , omit : ["closed"]
          }
      }
/* If the audit runtime-level is equal to 2, the audit covers methods
* of the session object in the following cases:
* 1.Rules of the session object return any result.
* 2.A query expression of the session object is executed. */
  , 2 :
      { session :
```

```
          { kss : ["granted", "denied"]
          }
      }
  }
```

Example 2

```
// Describing a test security audit profile
// base – Base security model object
// re – Regex security model object
audit profile test =
/* If the audit runtime-level is equal to 0, rules of the base object
 * and expressions of the re object are not covered by the audit. */
  { 0 :
      { base :
        { kss : []
        }
      , re :
        { kss : []
        , emit : []
        }
      }
/* If the audit runtime-level is equal to 1, rules of the
 * base object are not covered by the audit, and expressions of the
 * re object are covered by the audit.*/
  , 1 :
      { base :
        { kss : []
        }
      , re :
        { kss : ["granted"]
        , emit : ["match", "select"]
        }
      }
/* If the audit runtime-level is equal to 2, rules of the base object
 * and expressions of the re object are covered by the audit. Rules
 * of the base object are covered by the audit irrespective of the
 * result that they return.*/
  , 2 :
      { base :
        { kss : ["granted", "denied"]
        }
      , re :
        { kss : ["granted"]
        , emit : ["match", "select"]
        }
      }
  }
```

KasperskyOS security models

Auxiliary security models

The following security models are considered auxiliary:

- Pred – comparison operations
- Bool – logical operations

- Math – integer arithmetic operations, bitwise operations, and integer type casting operations
- Struct – access to structural data elements

The PSL file containing descriptions of auxiliary security models is located in the KasperskyOS SDK at the following path:

`toolchain/include/nk/basic.psl`

Pred security model

Pred security model object

The `basic.psl` file contains a declaration that creates a Pred security model object named `pred`. Consequently, inclusion of the `basic.psl` file into the solution security policy description will create a Pred security model object by default.

A Pred security model object does not have any parameters and cannot be covered by a security audit.

It is not necessary to create additional Pred security model objects.

Pred security model methods

A Pred security model contains expressions that perform comparison operations and return values of the `Boolean` type. To call these expressions, use the following comparison operators:

- `<ScalarLiteral> == <ScalarLiteral>` – "equals".
- `<ScalarLiteral> != <ScalarLiteral>` – "does not equal".
- `<Number> < <Number>` – "is less than".
- `<Number> <= <Number>` – "is less than or equal to".
- `<Number> > <Number>` – "is greater than".
- `<Number> >= <Number>` – "is greater than or equal to".

The Pred security model also contains the `empty` expression that determines whether data contains its own structural elements. This expression returns values of the `Boolean` type. If data does not contain its own structural elements (for example, a set is empty), the expression returns `true`, otherwise it returns `false`. To call the expression, use the following construct:

```
pred.empty (<Text | Set | List | Map>) | ()
```

Bool security model

Bool security model object

The `basic.psl` file contains a declaration that creates a Bool security model object named `bool`. Consequently, inclusion of the `basic.psl` file into the solution security policy description will create a Bool security model object by default.

A Bool security model object does not have any parameters and cannot be covered by a security audit.

It is not necessary to create additional Bool security model objects.

Bool security model methods

The Bool security model contains expressions that perform logical operations and return values of the `Boolean` type. To call these expressions, use the following logical operators:

- `! <Boolean>` – "logical NOT".
- `<Boolean> && <Boolean>` – "logical AND".
- `<Boolean> || <Boolean>` – "logical OR".
- `<Boolean> ==> <Boolean>` – "implication" (`! <Boolean> || <Boolean>`).

The Bool security model also contains the `all`, `any` and `cond` expressions.

The expression `all` performs a "logical AND" for an arbitrary number of values of `Boolean` type. It returns values of the `Boolean` type. It returns `true` if an empty list of values (`[]`) is passed via the parameter. To call the expression, use the following construct:

```
bool.all (<List<Boolean>>)
```

The expression `any` performs a "logical OR" for an arbitrary number of values of `Boolean` type. It returns values of the `Boolean` type. It returns `false` if an empty list of values (`[]`) is passed via the parameter. To call the expression, use the following construct:

```
bool.any (<List<Boolean>>)
```

`cond` expression performs a ternary conditional operation. Returns values of the `ScalarLiteral` type. To call the expression, use the following construct:

```
bool.cond
  { if   : <Boolean> // Condition
  , then : <ScalarLiteral> // Value returned when the condition is true
  , else : <ScalarLiteral> // Value returned when the condition is false
  }
```

In addition to expressions, the Bool security model includes the `assert` rule that works the same as the rule of the same name included in the Base security model.

Math security model

Math security model object

The `basic.psl` file contains a declaration that creates a Math security model object named `math`. Consequently, inclusion of the `basic.psl` file into the solution security policy description will create a Math security model object by default.

A Math security model object does not have any parameters and cannot be covered by a security audit.

It is not necessary to create additional Math security model objects.

Math security model methods

The Math security model contains expressions that perform integer arithmetic operations, bitwise operations, and integer type casting operations. To call a part of these expressions, use the following arithmetic operators:

- `<Number> + <Number>` – "addition". Returns values of the `Number` type.
- `<Number> - <Number>` – "subtraction". Returns values of the `Number` type.
- `<Number> * <Number>` – "multiplication". Returns values of the `Number` type.
- `<Number> | <Number>` – "bitwise OR". Returns values of the `Number` type.
- `<Number> & <Number>` – "bitwise AND". Returns values of the `Number` type.
- `~ <Number>` – "bitwise complement". Returns values of the `Number` type.

The other expressions are as follows:

- `neg (<Signed>)` – "change number sign". Returns values of the `Signed` type.
- `abs (<Signed> )` – "get module of number". Returns values of the `Signed` type.
- `sum (<List<Number>>)` – "add numbers from list". Returns values of the `Number` type. It returns `0` if an empty list of values (`[]`) is passed via the parameter.
- `product (<List<Number>>)` – "multiple numbers from list". Returns values of the `Number` type. It returns `1` if an empty list of values (`[]`) is passed via the parameter.
- `uint64 (<Unsigned>)` – "casting to type `UInt64`".
- `uint32 (<UInt8 | UInt16 | UInt32>)` – "casting to type `UInt32`".
- `uint16 (<UInt8 | UInt16>)` – "casting to type `UInt16`".
- `sint64 (<Signed | UInt8 | UInt16 | UInt32>)` – "casting to type `SInt64`".
- `sint32 (<SInt8 | SInt16 | SInt32 | UInt8 | UInt16>)` – "casting to type `SInt32`".

- `sint16 (<SInt8 | SInt16 | UInt8>)` – "casting to type `SInt16`".
- `signedToUInt64 (<Signed>)` – "casting to type `UInt64`". Negative numbers are converted to positive numbers by subtracting them from *2^64*. For example, *-1* is converted to *2^64-1*.
- `signedToUInt32 (<SInt8 | SInt16 | SInt32>)` – "casting to type `UInt32`". Negative numbers are converted to positive numbers by subtracting them from *2^32*. For example, *-3* is converted to *2^32-3*.
- `signedToUInt16 (<SInt8 | SInt16>)` – "casting to type `UInt16`". Negative numbers are converted to positive numbers by subtracting them from *2^16*. For example, *-5* is converted to *2^16-5*.
- `signedToUInt8 (<SInt8>)` – "casting to type `UInt8`". Negative numbers are converted to positive numbers by subtracting them from *2^8*. For example, *-7* is converted to *2^8-7*.

To call these expressions, use the following construct:

```
math.<expression name> (<parameter>)
```

Struct security model

Struct security model object

The `basic.psl` file contains a declaration that creates a Struct security model object named `struct`. Consequently, inclusion of the `basic.psl` file into the solution security policy description will create a Struct security model object by default.

A Struct security model object does not have any parameters and cannot be covered by a security audit.

It is not necessary to create additional Struct security model objects.

Struct security model methods

The Struct security model contains expressions that provide access to structural data elements. To call these expressions, use the following constructs:

- `<dictionary>.<field name>` – "get access to dictionary field". The type of returned data corresponds to the type of dictionary field.
- `<List | Set | Sequence | Array>.[<element number>]` – "get access to data element". The type of returned data corresponds to the type of elements. The numbering of elements starts with zero. When out of bounds of dataset, the expression terminates with an error and the Kaspersky Security Module returns the "denied" decision.
- `<HandleDesc>.handle` – "get SID". Returns values of the `Handle` type. (For details on the correlation between handles and SID values, see Resource Access Control).
- `<HandleDesc>.rights` – "get handle permissions mask". Returns values of the `UInt32` type.

Parameters of interface methods are saved in a special dictionary named `message`. To obtain access to an interface method parameter, use the following construct:

```
message.<interface method parameter name>
```

The parameter name is specified in accordance with the IDL description.

To obtain access to structural elements of parameters, use the constructs corresponding to expressions of the Struct security model.

> To use expressions of the Struct security model, the security event description must be sufficiently precise so that it corresponds to IPC messages of the same type (for more details, see Binding methods of security models to security events). IPC messages of this type must contain the defined parameters of the interface method, and the interface method parameters must contain the defined structural elements.

Precedence and associativity of auxiliary security model expressions

The precedence and associativity of expressions of the auxiliary security models Pred, Bool, Math and Struct are presented in the table below.

Precedence and associativity of auxiliary security model expressions

Expressions	Precedence	Associativity	Security model	Special considerations
`pred.empty`	1	No	Pred	N/A

`bool.all`
`bool.any`
`bool.cond`
	1	No	Bool	N/A

`math.<expression name>`	1	No	Math	N/A
Struct security model expressions	1	No	Struct	N/A
`~`	2	No	Math	N/A
`!`	2	No	Bool	N/A
`*`	3	Left	Math	N/A
`+` `-`	4	Left	Math	N/A

`<`
`>`
`<=`
`>=`
`==`
`!=`
	5	No	Pred	N/A

`&&`	6	Left	Bool	N/A
`||`	7	Left	Bool	N/A
`==>`	8	Right	Bool	N/A

Expressions	Precedence	Associativity	Security model	Special considerations
&	3*	Left	Math	The precedence is lower than 2 but is incomparable with the precedences 3-8, therefore parentheses must be used to define the order of operations when jointly used with expressions that have precedences 3-8.
|	4*	Left	Math	The precedence is lower than 3* but is incomparable with the precedences 3-8, therefore parentheses must be used to define the order of operations when jointly used with expressions that have precedences 3-8.

The expressions of all security models (not just auxiliary ones) have a precedence of 1.

Base security model

The Base security model implements basic logic.

A PSL file containing a description of the Base security model is located in the KasperskyOS SDK at the following path:

`toolchain/include/nk/base.psl`

Base security model object

The `base.psl` file contains a declaration that creates a Base security model object named `base`. Consequently, inclusion of the `base.psl` file into the solution security policy description will create a Base security model object by default. Methods of this object can be called without indicating the object name.

A Base security model object does not have any parameters.

A Base security model object can be covered by a security audit. There are no audit conditions specific to the Base security model.

It is necessary to create additional objects of the Base security model in the following cases:

- You need to configure a security audit differently for different objects of the Base security model (for example, you can apply different audit profiles or different audit configurations of the same profile for different objects).
- You need to distinguish between calls of methods provided by different objects of the Base security model (audit data includes the name of the security model method and the name of the object that provides this method, so you can verify that the method of a specific object was called).

Base security model methods

The Base security model contains the following rules:

- `grant ()`

  It has a parameter of the `()` type. It returns the "granted" result.

Example:

```
/* A client of the foo class is allowed
 * to query a server of the bar class. */
request src=foo dst=bar { grant () }
```

- `assert (<Boolean>)`

  It returns the "granted" result if the `true` value is passed via the parameter. Otherwise it returns the "denied" result.

  Example:

```
/* Any client in the solution will be allowed to query a server of the foo class
 * by calling the Send method of the net.Net endpoint if the port parameter
 * of the Send method will be used to pass a value greater than 80. Otherwise any
 * client in the solution will be prohibited from querying a server of the
 * foo class by calling the Send method of the net.Net endpoint. */
request dst=foo endpoint=net.Net method=Send { assert (message.port > 80) }
```

- `deny (<Boolean>) | ()`

  It returns the "denied" result if the `true` or `()` value is passed via the parameter. Otherwise it returns the "granted" result.

  Example:

```
/* A server of the foo class is not allowed to
 * respond to a client of the bar class. */
response src=foo dst=bar { deny () }
```

- `set_level (<UInt8>)`

  It sets the security audit runtime-level equal to the value passed via this parameter. It returns the "granted" result. (For more details about the security audit runtime-level, see Describing security audit profiles.)

  Example:

```
/* A process of the foo class will receive the "allowed" decision from the
 * Kaspersky Security Module if it calls the
 * SetAuditLevel security interface method to change the security audit runtime-level. */
security src=foo method=SetAuditLevel { set_level (message.audit_level) }
```

Regex security model

The Regex security model implements text data validation based on statically defined regular expressions.

A PSL file containing a description of the Regex security model is located in the KasperskyOS SDK at the following path:

`toolchain/include/nk/regex.psl`

Regex security model object

The `regex.psl` file contains a declaration that creates a Regex security model object named `re`. Consequently, inclusion of the `regex.psl` file into the solution security policy description will create a Regex security model object by default.

A Regex security model object does not have any parameters.

A Regex security model object can be covered by a security audit. In this case, you also need to define the audit conditions specific to the Regex security model. To do so, use the following constructs in the audit configuration description:

- `emit : ["match"]` – the audit is performed if the `match` method is called.
- `emit : ["select"]` – the audit is performed if the `select` method is called.
- `emit : ["match", "select"]` – the audit is performed if the `match` or `select` method is called.
- `emit : []` – the audit is not performed.

It is necessary to create additional objects of the Regex security model in the following cases:

- You need to configure a security audit differently for different objects of the Regex security model (for example, you can apply different audit profiles or different audit configurations of the same profile for different objects).
- You need to distinguish between calls of methods provided by different objects of the Regex security model (audit data includes the name of the security model method and the name of the object that provides this method, so you can verify that the method of a specific object was called).

Regex security model methods

The `Regex` security model contains the following expressions:

- `match {text : <Text>, pattern : <Text>}`

  It returns a value of the `Boolean` type. If the specified `text` matches the `pattern` regular expression, it returns `true`. Otherwise it returns `false`.

  Example:

  ```
  assert (re.match {text : message.text, pattern : "[0-9]*"})
  ```

- `select {text : <Text>}`

  It is intended to be used as an expression that verifies fulfillment of the conditions in the `choice` construct (for details on the `choice` construct, see Binding methods of security models to security events). It checks whether the specified `text` matches regular expressions. Depending on the results of this check, various options for security event handling can be performed.

Example:

```
choice (re.select {text : "hello world"}) {
    "hello\ .*": grant ()
    ".*world" : grant ()
    _         : deny ()
   }
```

Syntax of regular expressions of the Regex security model

A regular expression for the `match` method of the Regex security model can be written in two ways: within the multi-line `regex` block or as a text literal.

> When writing a regular expression as a text literal, all backslash instances must be doubled.

For example, the following two regular expressions are identical:

```
// Regular expression within the multi-line regex block
{ pattern:
    ```regex
    Hello\ world\!
    ```
, text: "Hello world!"
}
// Regular expression as a text literal (doubled backslash)
{ pattern: "Hello\\ world\\!"
, text: "Hello world!"
}
```

Regular expressions for the `select` method of the Regex security model are written as text literals with a double backslash.

A regular expression is defined as a template string and may contain the following:

- Literals (ordinary characters)
- Metacharacters (characters with special meanings)
- White-space characters
- Character sets
- Character groups
- Operators for working with characters

Regular expressions are case sensitive.

Literals and metacharacters in regular expressions

- A literal can be any ASCII character except the metacharacters `.()*&|!?+[]\` and a white-space character. (Unicode characters are not supported.)

  For example, the regular expression `KasperskyOS` corresponds to the text `KasperskyOS`.

- Metacharacters have special meanings that are presented in the table below.

  Special meanings of metacharacters

Metacharacter	Special meaning
`[]`	Square brackets (braces) denote the beginning and end of a set of characters.
`()`	Round brackets (parentheses) denote the beginning and end of a group of characters.
`*`	An asterisk denotes an operator indicating that the character preceding it can repeat zero or more times.
`+`	A plus sign denotes an operator indicating that the character preceding it can repeat one or more times.
`?`	A question mark denotes an operator indicating that the character preceding it can repeat zero or one time.
`!`	An exclamation mark denotes an operator excluding the subsequent character from the list of valid characters.
`|`	A vertical line denotes an operator for selection between characters (logically close to the "OR" conjunction).
`&`	An ampersand denotes an operator for overlapping of multiple conditions (logically close to the "AND" conjunction).
`.`	A dot denotes any character. For example, the regular expression `K.S` corresponds to the sequences of characters `KOS`, `KoS`, `KES` and a multitude of other sequences consisting of three characters that begin with `K` and end with `S`, and in which the second character can be any character: literal, metacharacter, or dot.
`\`	`\<metaSymbol>` A backslash indicates that the metacharacter that follows it will lose its special meaning and instead be interpreted as a literal. A backslash placed before a metacharacter is known as an escape character. For example, a regular expression that consists of a dot metacharacter (`.`) corresponds to any character. However, a regular expression that consists of a backslash with a dot (`\.`) corresponds to only a dot character. Accordingly, a backslash also escapes another subsequent backslash. For example, the regular expression `C:\\Users` corresponds to the sequence of characters `C:\Users`.

- The `^` and `$` characters are not used to designate the start and end of a line.

White-space characters in regular expressions

- A space character has an ASCII code of `20` in a hexadecimal number system and has an ASCII code of `40` in an octal number system. Although a space character does not infer any special meaning, it must be escaped to avoid any ambiguous interpretation by the regular expression interpreter.

  For example, the regular expression `Hello\ world` corresponds to the sequence of characters `Hello world`.

- `\r`

  Carriage return character.

- `\n`

  Line break character.

- `\t`

  Horizontal tab character.

Definition of a character based on its octal or hexadecimal code in regular expressions

- `\x{<hex>}`

  Definition of a character using its `hex` code from the ASCII character table. The character code must be less than `0x100`.

  For example, the regular expression `Hello\x{20}world` corresponds to the sequence of characters `Hello world`.

- `\o{<octal>}`

  Definition of a character using its `octal` code from the ASCII character table. The character code must be less than `0o400`.

  For example, the regular expression `\o{75}` corresponds to the `=` character.

Sets of characters in regular expressions

A character set is defined within square brackets `[]` as a list or range of characters. A character set tells the regular expression interpreter that only one of the characters listed in the set or range of characters can be at this specific location in a sequence of characters. A character set cannot be left blank.

- `[<BracketSpec>]` – character set.

  One character corresponds to any character from the `BracketSpec` character set.

  For example, the regular expression `K[OE]S` corresponds to the sequences of characters `KOS` and `KES`.

- `[^<BracketSpec>]` – inverted character set.

  One character corresponds to any character that is not in the `BracketSpec` character set.

  For example, the regular expression `K[^OE]S` corresponds to the sequences of characters `KAS`, `K8S` and any other sequences consisting of three characters that begin with `K` and end with `S`, excluding `KOS` and `KES`.

The `BracketSpec` character set can be listed explicitly or can be defined as a range of characters. When defining a range of characters, the first and last character in the set must be separated with a hyphen.

- `[<Digit1>-<DigitN>]`

  Any number from the range `Digit1`, `Digit2`, … ,`DigitN`.

  For example, the regular expression `[0-9]` corresponds to any numerical digit. The regular expressions `[0-9]` and `[0123456789]` are identical.

  Please note that a range is defined by one character before a hyphen and one character after the hyphen. The regular expression `[1-35]` corresponds only to the characters `1`, `2`, `3` and `5`, and does not represent the range of numbers from `1` to `35`.

- `[<Letter1>-<LetterN>]`

  Any English letter from the range `Letter1`, `Letter2`, … , `LetterN` (these letters must be in the same case).

  For example, the regular expression `[a-zA-Z]` corresponds to all letters in uppercase and lowercase from the ASCII character table.

The ASCII code for the upper boundary character of a range must be higher than the ASCII code for the lower boundary character of the range.

For example, the regular expressions `[5-2]` or `[z-a]` are invalid.

The hyphen (minus) `-` character is interpreted as a special character only within a set of characters. Outside of a character set, a hyphen is a literal. For this reason, the `\` metacharacter does not have to precede a hyphen. To use a hyphen as a literal within a character set, it must be indicated first or last in the set.

Examples:

The regular expressions `[-az]` and `[az-]` correspond to the characters `a`, `z` and `-`.

The regular expression `[a-z]` corresponds to any of the 26 English letters from `a` to `z` in lowercase.

The regular expression `[-a-z]` corresponds to any of the 26 English letters from `a` to `z` in lowercase and `-`.

The circumflex (caret character) `^` is interpreted as a special character only within a character set when it is located directly after an opening square bracket. Outside of a character set, a circumflex is a literal. For this reason, the `\` metacharacter does not have to precede a circumflex. To use a circumflex as a literal within a character set, it must be indicated in a location other than first in the set.

Examples:

The regular expression `[0^9]` correspond to the characters `0`, `9` and `^`.

The regular expression `[^09]` corresponds to any character except `0` and `9`.

Within a character set, the metacharacters `*.&|!?+` lose their special meaning and are instead interpreted as literals. Therefore, they do not have to be preceded by the `\` metacharacter. The backslash `\` retains its special meaning within a character set.

For example, the regular expressions `[a.]` and `[a\.]` are identical and correspond to the character `a` and a dot interpreted as a literal.

Groups of characters and operators in regular expressions

A character group uses parentheses `()` to distinguish its portion (subexpression) within a regular expression. Groups are normally used to allocate subexpressions as operands. Groups can be embedded into each other.

> Operators are applied to more than one character in a regular expression only if they are immediately before or after the definition of a set or group of characters. If this is the case, the operator is applied to the entire group or set of characters.

The syntax contains definitions of the following operators (listed in descending order of their priority):

- `!<Expression>`, where `Expression` can be a character, set or group of characters.

  This operator excludes the `Expression` from the list of valid expressions.

  Examples:

  The regular expression `K!OS` corresponds to the sequences of characters `KoS`, `KES`, and a multitude of other sequences that consist of three characters and begin with `K` and end with `S`, excluding `KOS`.

The regular expression `K!(OS)` corresponds to the sequences of characters `Kos`, `KES`, `KOT`, and a multitude of other sequences that consist of three characters and begin with `K`, excluding `KOS`.

The regular expression `K![OE]S` corresponds to the sequences of characters `KoS`, `KeS`, `K;S`, and a multitude of other sequences that consist of three characters and begin with `K` and end with `S`, excluding `KOS` and `KES`.

- `<Expression>*`, where `Expression` can be a character, set or group of characters.

  This operator means that the `Expression` may occur in the specific position zero or more times.

  Examples:

  The regular expression `0-9*` corresponds to the sequences of characters `0-`, `0-9`, `0-99`, … .

  The regular expression `(0-9)*` corresponds to the empty sequence `""` and the sequences of characters `0-9`, `0-90-9`, … .

  The regular expression `[0-9]*` corresponds to the empty sequence `""` and any non-empty sequence of numbers.

- `<Expression>+`, where `Expression` can be a character, set or group of characters.

  This operator means that the `Expression` may occur in the specific position one or more times.

  Examples:

  The regular expression `0-9+` corresponds to the sequences of characters `0-9`, `0-99`, `0-999`, … .

  The regular expression `(0-9)+` corresponds to the sequences of characters `0-9`, `0-90-9`, … .

  The regular expression `[0-9]+` corresponds to any non-empty sequence of numbers.

- `<Expression>?`, where `Expression` can be a character, set or group of characters.

  This operator means that the `Expression` may occur in the specific position zero or one time.

  Examples:

  The regular expression `https?://` corresponds to the sequences of characters `http://` and `https://`.

  The regular expression `K(aspersky)?OS` corresponds to the sequences of characters `KOS` and `KasperskyOS`.

- `<Expression1><Expression2>` – concatenation. `Expression1` and `Expression2` can be characters, sets or groups of characters.

  This operator does not have a specific designation. In the resulting expression, `Expression2` follows `Expression1`.

  For example, concatenation of the sequences of characters `micro` and `kernel` will result in the sequence of characters `microkernel`.

- `<Expression1>|<Expression2>` – disjunction. `Expression1` and `Expression2` can be characters, sets or groups of characters.

  This operator selects either `Expression1` or `Expression2`.

  Examples:

  The regular expression `KO|ES` corresponds to the sequences of characters `KO` and `ES`, but not `KOS` or `KES` because the concatenation operator has a higher priority than the disjunction operator.

  The regular expression `Press (OK|Cancel)` corresponds to the sequences of characters `Press OK` or `Press Cancel`.

  The regular expression `[0-9]|()` corresponds to numbers from `0` to `9` or an empty string.

- `<Expression1>&<Expression2>` – conjunction. `Expression1` and `Expression2` can be characters, sets or groups of characters.

  This operator intersects the result of `Expression1` with the result of `Expression2`.

Examples:

The regular expression `[0-9]&[^3]` corresponds to numbers from `0` to `9`, excluding `3`.

The regular expression `[a-zA-Z]&()` corresponds to all English letters and an empty string.

HashSet security model

The HashSet security model associates resources with one-dimensional tables of unique values of the same type, adds or deletes these values, and checks whether a defined value is in the table. For example, a process of the network server can be associated with the set of ports that this server is allowed to open. This association can be used to check whether the server is allowed to initiate the opening of a port.

A PSL file containing a description of the HashSet security model is located in the KasperskyOS SDK at the following path:

`toolchain/include/nk/hashmap.psl`

HashSet security model object

To use the HashSet security model, you need to create an object or objects of this model.

A HashSet security model object contains a pool of one-dimensional tables of the same size intended for storing the values of one type. A resource can be associated with only one table from the tables pool of each HashSet security model object.

A HashSet security model object has the following parameters:

- `type Entry` – type of values in tables (these can be integer types, `Boolean` type, and dictionaries and tuples based on integer types and the `Boolean` type).
- `config` – configuration of the pool of tables:
  - `set_size` – size of the table.
  - `pool_size` – number of tables in the pool.

All parameters of a HashSet security model object are required.

Example:

```
policy object s : HashSet {
    type Entry = UInt32

    config =
      { set_size : 5
      , pool_size : 2
      }
}
```

A HashSet security model object can be covered by a security audit. There are no audit conditions specific to the HashSet security model.

It is necessary to create multiple objects of the HashSet security model in the following cases:

- You need to configure a security audit differently for different objects of the HashSet security model (for example, you can apply different audit profiles or different audit configurations of the same profile for different objects).
- You need to distinguish between calls of methods provided by different objects of the HashSet security model (audit data includes the name of the security model method and the name of the object that provides this method, so you can verify that the method of a specific object was called).
- You need to use tables of different sizes and/or with different types of values.

HashSet security model init rule

```
init {sid : <Sid>}
```

It associates a free table from the tables pool with the `sid` resource. If the free table contains values after its previous use, these values are deleted.

It returns the "allowed" result if an association was created between the table and the `sid` resource.

It returns the "denied" result in the following cases:

- There are no free tables in the pool.
- The `sid` resource is already associated with a table from the tables pool of the HashSet security model object being used.
- The `sid` value is outside of the permissible range.

Example:

```
/* A process of the Server class will be allowed to start if,
 * at startup initiation, an association will be created
 * between this process and the table. Otherwise the startup of a process of the
 * Server class will be denied. */
execute dst=Server {
    s.init {sid : dst_sid}
}
```

HashSet security model fini rule

```
fini {sid : <Sid>}
```

It deletes the association between the table and the `sid` resource (the table becomes free).

It returns the "allowed" result if the association between the table and the `sid` resource was deleted.

It returns the "denied" result in the following cases:

- The `sid` resource is not associated with a table from the tables pool of the HashSet security model object being used.
- The `sid` value is outside of the permissible range.

HashSet security model add rule

```
add {sid : <Sid>, entry : <Entry>}
```

It adds the `entry` value to the table associated with the `sid` resource.

It returns the "allowed" result in the following cases:

- The rule added the `entry` value to the table associated with the `sid` resource.
- The table associated with the `sid` resource already contains the `entry` value.

It returns the "denied" result in the following cases:

- The table associated with the `sid` resource is completely full.
- The `sid` resource is not associated with a table from the tables pool of the HashSet security model object being used.
- The `sid` value is outside of the permissible range.

Example:

```
/* A process of the Server class will receive the "allowed" decision from
 * the Kaspersky Security Module by calling the
 * Add security interface method if, when this method is called, the value
 * 5 will be added to the table associated with this
 * process, or is already in the table. Otherwise
 * a process of the Server class will receive the "denied" decision from the
 * security module by calling the
 * Add security interface method. */
security src=Server, method=Add {
    s.add {sid : src_sid, entry : 5}
}
```

HashSet security model remove rule

```
remove {sid : <Sid>, entry : <Entry>}
```

It deletes the `entry` value from the table associated with the `sid` resource.

It returns the "allowed" result in the following cases:

- The rule deleted the `entry` value from the table associated with the `sid` resource.
- The table associated with the `sid` resource does not contain the `entry` value.

It returns the "denied" result in the following cases:

- The `sid` resource is not associated with a table from the tables pool of the HashSet security model object being used.
- The `sid` value is outside of the permissible range.

HashSet security model contains expression

```
contains {sid : <Sid>, entry : <Entry>}
```

It checks whether the `entry` value is in the table associated with the `sid` resource.

It returns a value of the `Boolean` type. If the `entry` value is in the table associated with the `sid` resource, it returns `true`. Otherwise it returns `false`.

It runs incorrectly in the following cases:

- The `sid` resource is not associated with a table from the tables pool of the HashSet security model object being used.
- The `sid` value is outside of the permissible range.

When the expression runs incorrectly, the Kaspersky Security Module returns the "denied" decision.

Example:

```
/* A process of the Server class will receive the "allowed" decision from
 * the Kaspersky Security Module by calling the
 * Check security interface method if the value 42 is in the table
 * associated with this process. Otherwise a process of the
 * Server class will receive the "denied" decision from the security module
 * by calling the Check security interface method. */
security src=Server, method=Check {
    assert(s.contains {sid : src_sid, entry : 42})
}
```

StaticMap security model

The StaticMap security model associates resources with two-dimensional "key–value" tables, reads and modifies the values of keys. For example, a process of the driver can be associated with the MMIO memory region that this driver is allowed to use. This will require two keys whose values define the base address and the size of the MMIO memory region. This association can be used to check whether the driver can query the MMIO memory region that it is attempting to access.

Keys in the table have the same type but are unique and immutable. The values of keys in the table have the same type.

There are two simultaneous instances of the table: base instance and working instance. Both instances are initialized by the same data. Changes are made first to the working instance and then can be added to the base instance, or vice versa: the working instance can be changed by using previous values from the base instance. The values of keys can be read from the base instance or working instance of the table.

A PSL file containing a description of the StaticMap security model is located in the KasperskyOS SDK at the following path:

`toolchain/include/nk/staticmap.psl`

StaticMap security model object

To use the StaticMap security model, you need to create an object or objects of this model.

A StaticMap security model object contains a pool of two-dimensional "key–value" tables that have the same size. A resource can be associated with only one table from the tables pool of each StaticMap security model object.

A StaticMap security model object has the following parameters:

- `type Value` – type of values of keys in tables (integer types are supported).
- `config` – configuration of the pool of tables:
  - `keys` – table containing keys and their default values (keys have the `Key = Text | List<UInt8>` type).
  - `pool_size` – number of tables in the pool.

All parameters of a StaticMap security model object are required.

Example:

```
policy object m : StaticMap {
    type Value = UInt16

    config =
      { keys:
        { "k1" : 0
        , "k2" : 1
        }
      , pool_size : 2
      }
}
```

A StaticMap security model object can be covered by a security audit. There are no audit conditions specific to the StaticMap security model.

It is necessary to create multiple objects of the StaticMap security model in the following cases:

- You need to configure a security audit differently for different objects of the StaticMap security model (for example, you can apply different audit profiles or different audit configurations of the same profile for different objects).
- You need to distinguish between calls of methods provided by different objects of the StaticMap security model (audit data includes the name of the security model method and the name of the object that provides this method, so you can verify that the method of a specific object was called).
- You need to use tables with different sets of keys and/or different types of key values.

StaticMap security model init rule

```
init {sid : <Sid>}
```

It associates a free table from the tables pool with the `sid` resource. Keys are initialized by the default values.

It returns the "allowed" result if an association was created between the table and the `sid` resource.

It returns the "denied" result in the following cases:

- There are no free tables in the pool.
- The `sid` resource is already associated with a table from the tables pool of the StaticMap security model object being used.
- The `sid` value is outside of the permissible range.

Example:

```
/* A process of the Server class will be allowed to start if,
 * at startup initiation, an association will be created
 * between this process and the table. Otherwise the startup of a process of the
 * Server class will be denied. */
execute dst=Server {
    m.init {sid : dst_sid}
}
```

StaticMap security model fini rule

```
fini {sid : <Sid>}
```

It deletes the association between the table and the `sid` resource (the table becomes free).

It returns the "allowed" result if the association between the table and the `sid` resource was deleted.

It returns the "denied" result in the following cases:

- The `sid` resource is not associated with a table from the tables pool of the StaticMap security model object being used.
- The `sid` value is outside of the permissible range.

StaticMap security model set rule

```
set {sid : <Sid>, key : <Key>, value : <Value>}
```

It assigns the specified `value` to the specified `key` in the working instance of the table associated with the `sid` resource.

It returns the "allowed" result if the specified `value` was assigned to the specified `key` in the working instance of the table associated with the `sid` resource. (The current value of the key will be overwritten even if it is equal to the new value.)

It returns the "denied" result in the following cases:

- The specified `key` is not in the table associated with the `sid` resource.
- The `sid` resource is not associated with a table from the tables pool of the StaticMap security model object being used.
- The `sid` value is outside of the permissible range.

Example:

```
/* A process of the Server class will receive the "allowed" decision from
 * the Kaspersky Security Module by calling the
 * Set security interface method if, when this method is called, the value 2
 * will be assigned to key k1 in the working instance of the table
 * associated with this process. Otherwise a process of the
 * Server class will receive the "denied" decision from the security module
 * by calling the Set security interface method. */
security src=Server, method=Set {
    m.set {sid : src_sid, key : "k1", value : 2}
}
```

StaticMap security model commit rule

```
commit {sid : <Sid>}
```

It copies the values of keys from the working instance to the base instance of the table associated with the `sid` resource.

It returns the "allowed" result if the values of keys were copied from the working instance to the base instance of the table associated with the `sid` resource.

It returns the "denied" result in the following cases:

- The `sid` resource is not associated with a table from the tables pool of the StaticMap security model object being used.
- The `sid` value is outside of the permissible range.

StaticMap security model rollback rule

```
rollback {sid : <Sid>}
```

It copies the values of keys from the base instance to the working instance of the table associated with the `sid` resource.

It returns the "allowed" result if the values of keys were copied from the base instance to the working instance of the table associated with the `sid` resource.

It returns the "denied" result in the following cases:

- The `sid` resource is not associated with a table from the tables pool of the StaticMap security model object being used.
- The `sid` value is outside of the permissible range.

StaticMap security model get expression

```
get {sid : <Sid>, key : <Key>}
```

It returns the value of the specified `key` from the base instance of the table associated with the `sid` resource.

It returns a value of the `Value` type.

It runs incorrectly in the following cases:

- The specified `key` is not in the table associated with the `sid` resource.
- The `sid` resource is not associated with a table from the tables pool of the StaticMap security model object being used.
- The `sid` value is outside of the permissible range.

When the expression runs incorrectly, the Kaspersky Security Module returns the "denied" decision.

Example:

```
/* A process of the Server class will receive the "allowed" decision from
 * the Kaspersky Security Module by calling the
 * Get security interface method if the value of key k1 in the base
 * instance of the table associated with this process
 * is not zero. Otherwise a process of the Server class will receive
 * the "denied" decision from the security module
 * by calling the Get security interface method. */
```

```
security src=Server, method=Get {
    assert(m.get {sid : src_sid, key : "k1"} != 0)
}
```

StaticMap security model get_uncommitted expression

```
get_uncommitted {sid: <Sid>, key: <Key>}
```

It returns the value of the specified `key` from the working instance of the table associated with the `sid` resource.

It returns a value of the `Value` type.

It runs incorrectly in the following cases:

- The specified `key` is not in the table associated with the `sid` resource.
- The `sid` resource is not associated with a table from the tables pool of the StaticMap security model object being used.
- The `sid` value is outside of the permissible range.

When the expression runs incorrectly, the Kaspersky Security Module returns the "denied" decision.

StaticMap security model initialized expression

```
initialized {sid: <Sid>}
```

The expression checks whether the `sid` resource is associated with a specific table from the pool of tables of the StaticMap object being used. It returns `true` if the `sid` resource is associated with the table (initialized), otherwise it returns `false`.

Usage requirements:

- The expression can only be used within a `choice` construct as a security model expression that verifies the fulfillment of conditions. (For more information about the `choice` construct, see the "Embedded constructs for binding methods of security models to security events" subsection of the "Binding methods of security models to security events" section.)
- Within the `choice` construct, you must specify a conditional section if none of the conditions are met (`_`).

Example:

```
execute {
  // The expression will return false if dst_sid is not associated with the table, otherwise it will return true.
  choice(counters.initialized {sid: dst_sid}) {
    // Associate the sid resource with the table
    false : counters.init {sid: dst_sid}
    // Otherwise, return the "denied" decision
    _ : deny ()
  }
}
```

See also:

PSL data types

StaticMap security model init rule

Flow security model

The Flow security model associates resources with finite-state machines, receives and modifies the states of finite-state machines, and checks whether the state of the finite-state machine is within the defined set of states. For example, a process can be associated with a finite-state machine to allow or prohibit this process from using storage and/or the network depending on the state of the finite-state machine.

A PSL file containing a description of the Flow security model is located in the KasperskyOS SDK at the following path:

`toolchain/include/nk/flow.psl`

Flow security model object

To use the Flow security model, you need to create an object or objects of this model.

One Flow security model object associates a set of resources with a set of finite-state machines that have the same configuration. A resource can be associated with only one finite-state machine of each Flow security model object.

A Flow security model object has the following parameters:

- `type State` – type that determines the set of states of the finite-state machine (variant type that combines text literals).
- `config` – configuration of the finite-state machine:
  - `states` – set of states of the finite-state machine (must match the set of states defined by the `State` type).
  - `initial` – initial state of the finite-state machine.
  - `transitions` – description of the permissible transitions between states of the finite-state machine.

All parameters of a Flow security model object are required.

Example:

```
policy object service_flow : Flow {
    type State = "sleep" | "started" | "stopped" | "finished"

    config = { states      : ["sleep", "started", "stopped", "finished"]
             , initial     : "sleep"
             , transitions : { "sleep"   : ["started"]
                             , "started" : ["stopped", "finished"]
                             , "stopped" : ["started", "finished"]
                             }
```

```
            }
}
```

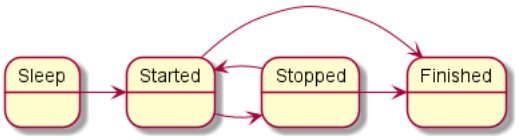


Diagram of finite-state machine states in the example

A Flow security model object can be covered by a security audit. You can also define the audit conditions specific to the Flow security model. To do so, use the following construct in the audit configuration description:

`omit : [<"state 1">[, ...]]` – the audit is not performed if the finite-state machine is in one of the listed states.

It is necessary to create multiple objects of the Flow security model in the following cases:

- You need to configure a security audit differently for different objects of the Flow security model (for example, you can apply different audit profiles or different audit configurations of the same profile for different objects).
- You need to distinguish between calls of methods provided by different objects of the Flow security model (audit data includes the name of the security model method and the name of the object that provides this method, so you can verify that the method of a specific object was called).
- You need to use finite-state machines with different configurations.

Flow security model init rule

```
init {sid : <Sid>}
```

It creates a finite-state machine and associates it with the `sid` resource. The created finite-state machine has the configuration defined in the settings of the Flow security model object being used.

It returns the "granted" result if an association was created between the finite-state machine and the `sid` resource.

It returns the "denied" result in the following cases:

- The `sid` resource is already associated with a finite-state machine of the Flow security model object being used.
- The `sid` value is outside of the permissible range.

Example:

```
/* A process of the Server class will be allowed to start
 * if, at startup initiation, an association will be created
 * between this process and the finite-state machine.
 * Otherwise the startup of the Server-class process will be denied. */
execute dst=Server {
    service_flow.init {sid : dst_sid}
}
```

Flow security model fini rule

```
fini {sid : <Sid>}
```

It deletes the association between the finite-state machine and the `sid` resource. The finite-state machine that is no longer associated with the resource is destroyed.

It returns the "granted" result if the association between the finite-state machine and the `sid` resource was deleted.

It returns the "denied" result in the following cases:

- The `sid` resource is not associated with a finite-state machine of the Flow security model object being used.
- The `sid` value is outside of the permissible range.

Flow security model enter rule

```
enter {sid : <Sid>, state : <State>}
```

It switches the finite-state machine associated with the `sid` resource to the specified `state`.

It returns the "granted" result if the finite-state machine associated with the `sid` resource was switched to the specified `state`.

It returns the "denied" result in the following cases:

- The transition to the specified `state` from the current state is not permitted by the configuration of the finite-state machine associated with the `sid` resource.
- The `sid` resource is not associated with a finite-state machine of the Flow security model object being used.
- The `sid` value is outside of the permissible range.

Example:

```
/* Any client in the solution will be allowed to query
 * a server of the Server class if the finite-state machine
 * associated with this server will be switched to
 * the "started" state when initiating the query. Otherwise
 * any client in the solution will be denied to query
 * a server of the Server class. */
request dst=Server {
    service_flow.enter {sid : dst_sid, state : "started"}
}
```

Flow security model allow rule

```
allow {sid : <Sid>, states : <Set<State>>}
```

It verifies that the state of the finite-state machine associated with the `sid` is in the set of defined `states`.

It returns the "granted" result if the state of the finite-state machine associated with the `sid` resource is in the set of defined `states`.

It returns the "denied" result in the following cases:

- The state of the finite-state machine associated with the `sid` resource is not in the set of defined `states`.
- The `sid` resource is not associated with a finite-state machine of the Flow security model object being used.
- The `sid` value is outside of the permissible range.

Example:

```
/* Any client in the solution is allowed to query a server
 * of the Server class if the finite-state machine associated with this server
 * is in the started or stopped state. Otherwise any client
 * in the solution will be prohibited from querying a server of the Server class. */
request dst=Server {
    service_flow.allow {sid : dst_sid, states : ["started", "stopped"]}
}
```

Flow security model query expression

```
query {sid : <Sid>}
```

It is intended to be used as an expression that verifies fulfillment of the conditions in the `choice` construct (for details on the `choice` construct, see Binding methods of security models to security events). It checks the state of the finite-state machine associated with the `sid` resource. Depending on the results of this check, various options for security event handling can be performed.

It runs incorrectly in the following cases:

- The `sid` resource is not associated with a finite-state machine of the Flow security model object being used.
- The `sid` value is outside of the permissible range.

When the expression runs incorrectly, the Kaspersky Security Module returns the "denied" decision.

Example:

```
/* Any client in the solution is allowed to query
 * a server of the ResourceDriver class if the finite-state machine
 * associated with this server is in the
 * "started" or "stopped" state. Otherwise any client in the solution
 * is prohibited from querying a server of the ResourceDriver class. */
request dst=ResourceDriver {
```

```
    choice (service_flow.query {sid : dst_sid}) {
        "started"   : grant ()
        "stopped"   : grant ()
        _           : deny ()
    }
}
```

Mic security model

The Mic security model implements mandatory integrity control. In other words, this security model provides the capability to manage data streams between different processes and between processes and the KasperskyOS kernel by controlling the integrity levels of processes, the kernel, and resources that are used via IPC.

> In Mic security model terminology, processes and the kernel are called subjects while resources are called objects. However, the information provided in this section slightly deviates from the terminology of the Mic security model. In this section, the term "object" is not used to refer to a "resource".

Data streams are generated between subjects when the subjects interact via IPC.

The *integrity level of a subject/resource* is the level of trust afforded to the subject/resource. The degree of trust in a subject depends on whether the subject interacts with untrusted external software/hardware systems or whether the subject has a proven quality level, for example. (The kernel has a high level of integrity.) The degree of trust in a resource depends on whether this resource was created by a trusted subject within a software/hardware system running KasperskyOS or if it was received from an untrusted external software/hardware system, for example.

The Mic security model is characterized by the following provisions:

- By default, data streams from subjects with less integrity to subjects with higher integrity are prohibited. You have the option of permitting such data streams if you can guarantee that the subjects with higher integrity will not be compromised.
- A resource consumer is prohibited from writing data to a resource if the integrity level of the resource is higher than the integrity level of the resource consumer.
- By default, a resource consumer is prohibited from reading data from a resource if the integrity level of the resource is lower than the integrity level of the resource consumer. You have the option to allow the resource consumer to perform such an operation if you can guarantee that the resource consumer will not be compromised.

Methods of a Mic security model let you perform the following operations:

- Assign integrity levels to subjects and resources.
- Unassign the integrity level from resources.
- Verify the permissibility of data streams based on a comparison of integrity levels.
- Increase the integrity levels of resources.

A PSL file containing a description of the Mic security model is located in the KasperskyOS SDK at the following path:

`toolchain/include/nk/mic.psl`

For an example of using the Mic security model, we can examine a secure software update for a software/hardware system running KasperskyOS. Four processes are involved in the update:

- `Downloader` is a low-integrity process that downloads a low-integrity update image from a remote server on the Internet.
- `Verifier` is a high-integrity process that verifies the digital signature of the low-integrity update image (high-integrity process that can read data from a low-integrity resource).
- `FileSystem` is a high-integrity process that manages the file system.
- `Updater` is a high-integrity process that applies an update.

A software update is performed according to the following scenario:

1. The `Downloader` downloads an update image and saves it to a file by transferring the contents of the image to the `FileSystem`. A low integrity level is assigned to this file.
2. The `Verifier` receives the update image from the `FileSystem` by reading the high-integrity file, and verifies its digital signature. If the signature is correct, the `Verifier` queries the `FileSystem` so that the `FileSystem` creates a copy of the file containing the update image. A high integrity level is assigned to the new file.
3. The `Updater` receives the update image from the `FileSystem` by reading the high-integrity file, and applies the update.

In this example, the Mic security model ensures that the high-integrity `Updater` process can read data only from a high-integrity update image. As a result, the update can be applied only after the digital signature of the update image is verified.

Mic security model object

To use the Mic security model, you need to create an object or objects of this model. You also need to assign a set of integrity levels for subjects and resources.

A Mic security model object has the following parameters:

- `config` – set of integrity levels or configuration of a set of integrity levels:
  - `degrees` – set of gradations for generating a set of integrity levels.
  - `categories` – set of categories for generating a set of integrity levels.

Examples:

```
policy object mic : Mic {
    config = ["LOW", "MEDIUM", "HIGH"]
}

policy object mic_po : Mic {
    config =
      { degrees    : ["low", "high"]
      , categories : ["net", "log"]
```

```
        }
}
```

A set of integrity levels is a partially ordered set that is linearly ordered or contains incomparable elements. The set {LOW, MEDIUM, HIGH} is linearly ordered because all of its elements are comparable to each other. Incomparable elements arise when a set of integrity levels is defined through a set of gradations and a set of categories. In this case, the set of integrity levels *L* is a Cartesian product of the Boolean set of categories *C* multiplied by the set of gradations *D*:

$$L = 2^C \times D.$$

The `degrees` and `categories` parameters in this example define the following set:

{

{}/low, {}/high,

{net}/low, {net}/high,

{log}/low, {log}/high,

{net,log}/low, {net,log}/high

}

In this set, {} means an empty set.

The order relation between elements of the set of integrity levels *L* is defined as follows:

$$l_i = A/B,$$

$$l_j = E/F,$$

$$l_i \prec l_j \Leftrightarrow \begin{cases} A \subseteq E, \\ B \leq F. \end{cases}$$

According to this order relation, the *j*th element exceeds the *i*th element if the subset of categories *E* includes the subset of categories *A*, and gradation *F* is greater than or equal to gradation *B*. Examples of comparing elements of the set of integrity levels *L*:

- The {net,log}/high element exceeds the {log}/low element because the "high" gradation is greater than the "low" gradation, and the subset of categories {net,log} includes the subset of categories {log}.
- The {net,log}/low element exceeds the {log}/low element because the levels of gradations for these elements are equal, and the subset of categories {net,log} includes the subset of categories {log}.
- The {net,log}/high element is the highest because it exceeds all other elements.
- The {}/low element is the lowest because all other elements exceed this element.

- The {net}/low and {log}/high elements are incomparable because the "high" gradation is greater than the "low" gradation but the subset of categories {log} does not include the subset of categories {net}.
- The {net,log}/low and {log}/high elements are incomparable because the "high" gradation is greater than the "low" gradation but the subset of categories {log} does not include the subset of categories {net,log}.

> For subjects and resources that have incomparable integrity levels, the Mic security model provides conditions that are analogous to the conditions that the security model provides for subjects and resources that have comparable integrity levels.
> By default, data streams between subjects that have incomparable integrity levels are prohibited. However, you have the option to allow such data streams if you can guarantee that the subjects receiving data will not be compromised. A resource consumer is prohibited from writing data to a resource and read data from a resource if the integrity level of the resource is incomparable to the integrity level of the resource consumer. You have the option to allow the resource consumer to read data from a resource if you can guarantee that the resource consumer will not be compromised.

A Mic security model object can be covered by a security audit. There are no audit conditions specific to the Mic security model.

It is necessary to create multiple objects of the Mic security model in the following cases:

- You need to configure a security audit differently for different objects of the Mic security model (for example, you can apply different audit profiles or different audit configurations of the same profile for different objects).
- You need to distinguish between calls of methods provided by different objects of the Mic security model (audit data includes the name of the security model method and the name of the object that provides this method, so you can verify that the method of a specific object was called).
- You need to use multiple variants of mandatory integrity control that may have different sets of integrity levels for subjects and resources, for example.

Mic security model create rule

```
create { source    : <Sid>
       , target    : <Sid>
       , container : <Sid | ()>
       , driver    : <Sid>
       , level     : <Level | ... | ()>
       }
```

Assign the specified integrity `level` to the `target` resource in the following situation:

- The `source` process initiates creation of the `target` resource.
- The `target` resource is managed by the `driver` subject, which is the resource provider or the KasperskyOS kernel.
- The `container` resource is a container for the `target` resource (for example, a directory is a container for files and/or other directories).

If the `container` field has the value `()`, the `target` resource is considered to be the root resource, which means that it has no container.

To define the integrity `level`, values of the `Level` type are used:

```
type Level = LevelFull | LevelNoCategory

type LevelFull =
  { degree     : Text | ()
  , categories : List<Text> | ()
  }

type LevelNoCategory = Text
```

The rule returns the "granted" result if a specific integrity `level` was assigned to the `target` resource.

The rule returns the "denied" result in the following cases:

- The `level` value exceeds the integrity level of the `source` process, `driver` subject or `container` resource.
- The `level` value is incomparable to the integrity level of the `source` process, `driver` subject or `container` resource.
- An integrity level was not assigned to the `source` process, `driver` subject, or `container` resource.
- The value of `source`, `target`, `container` or `driver` is outside of the permissible range.

Example:

```
/* A server of the updater.Realmserv class will be allowed to respond to
 * queries of any client in the solution calling the resolve method
 * of the realm.Reader endpoint if the resource whose creation is requested
 * by the client will be assigned the LOW integrity level during response initiation.
 * Otherwise a server of the updater.Realmserv class will be prohibited from responding to
 * queries of any client calling the resolve method of the realm.Reader endpoint. */
response src=updater.Realmserv,
         endpoint=realm.Reader {
             match method=resolve {
                 mic.create { source : dst_sid
                            , target : message.handle.handle
                            , container : ()
                            , driver : src_sid
                            , level : "LOW"
                            }
             }
}
```

Mic security model delete rule

```
delete { source    : <Sid>
       , target    : <Sid>
       , container : <Sid | ()>
       , driver    : <Sid>
       }
```

Unassigns the integrity level from the `target` resource in the following situation:

- The `source` process initiates deletion of the `target` resource.
- The `target` resource is managed by the `driver` subject, which is the resource provider or the KasperskyOS kernel.
- The `container` resource is a container for the `target` resource (for example, a directory is a container for files and/or other directories).

If the `container` field has the value `()`, the `target` resource is considered to be the root resource, which means that it has no container.

The rule returns the "granted" result if it unassigned the integrity level from the `target` resource.

The rule returns the "denied" result in the following cases:

- The integrity level of the `target` resource exceeds the integrity level of the `source` process or `driver` subject.
- The integrity level of the `target` resource is incomparable to the integrity level of the `source` process or `driver` subject.
- An integrity level was not assigned to the `source` process, `driver` subject, `target` resource or `container` resource.
- The value of `source`, `target`, `container` or `driver` is outside of the permissible range.

Example:

```
/* Any client in the solution will be allowed to query a server of the foo class
 * updater.Realmserv class by calling the del method of the realm.Reader endpoint if the
 * integrity level will be unassigned from the resource whose deletion is requested by the client.
 * Otherwise, any client in the solution will be prohibited from querying a server of the
 * updater.Realmserv class by calling the del method of the realm.Reader endpoint. */
request dst=updater.Realmserv,
        endpoint=realm.Reader {
            match method=del {
                mic.delete { source : src_sid
                           , target : message.handle.handle
                           , container : ()
                           , driver : dst_sid
                           }
              }
}
```

Mic security model execute rule

```
execute <ExecuteImage | ExecuteLevel>

type ExecuteImage =
  { image  : Sid
  , target : Sid
  , level  : Level | ... | ()
  , levelR : Level | ... | ()
  }

type ExecuteLevel =
  { image  : Sid | ()
  , target : Sid
  , level  : Level | ...
```

```
  , levelR : Level | ... | ()
  }
```

This assigns the specified integrity `level` to the `target` subject and defines the minimum integrity level of subjects and resources from which this subject can receive data (`levelR`). The code of the `target` subject is in the `image` executable file.

If the `level` field has the value `()`, the integrity level of the `image` executable file is assigned to the `target` subject. If the `image` field has the value `()`, the `level` field must have a value other than `()`.

If the `levelR` field has the value `()`, the `levelR` integrity level is assumed to be equal to the integrity level of the `target` subject.

To define the integrity `level` and `levelR`, values of the `Level` type are used. For the definition of the `Level` type, see Mic security model create rule.

The rule returns the "granted" result if it assigned the specified integrity `level` to the `target` subject and defined the minimum integrity level of subjects and resources from which this subject can receive data (`levelR`).

The rule returns the "denied" result in the following cases:

- The `level` value exceeds the integrity level of the `image` executable file.
- The `level` value is incomparable to the integrity level of the `image` executable file.
- The value of `levelR` exceeds the value of `level`.
- The `level` and `levelR` values are incomparable.
- An integrity level was not assigned to the `image` executable file.
- The `image` or `target` value is outside of the permissible range.

Example:

```
/* A process of the updater.Manager class will be allowed to start
 * if, at startup initiation, this process will be assigned
 * the integrity level LOW, and the minimum
 * integrity level will be defined for the processes and resources from which this
 * process can received data (LOW). Otherwise the startup of a process
 * of the updater.Manager class will be denied. */
execute src=Einit, dst=updater.Manager, method=main {
    mic.execute { target : dst_sid
                , image : ()
                , level : "LOW"
                , levelR : "LOW"
                }
}
```

Mic security model upgrade rule

```
upgrade { source    : <Sid>
        , target    : <Sid>
        , container : <Sid | ()>
        , driver    : <Sid>
        , level     : <Level | ...>
        }
```

This elevates the previously assigned integrity level of the `target` resource to the specified `level` in the following situation:

- The `source` process initiates elevation of the integrity level of the `target` resource.
- The `target` resource is managed by the `driver` subject, which is the resource provider or the KasperskyOS kernel.
- The `container` resource is a container for the `target` resource (for example, a directory is a container for files and/or other directories).

If the `container` field has the value `()`, the `target` resource is considered to be the root resource, which means that it has no container.

To define the integrity `level`, values of the `Level` type are used. For the definition of the `Level` type, see Mic security model create rule.

The rule returns the "granted" result if it elevated the previously assigned integrity level of the `target` resource to the `level` value.

The rule returns the "denied" result in the following cases:

- The `level` value does not exceed the integrity level of the `target` resource.
- The `level` value exceeds the integrity level of the `source` process, `driver` subject or `container` resource.
- The integrity level of the `target` resource exceeds the integrity level of the `source` process.
- An integrity level was not assigned to the `source` process, `driver` subject, or `container` resource.
- The value of `source`, `target`, `container` or `driver` is outside of the permissible range.

Mic security model call rule

```
call {source : <Sid>, target : <Sid>}
```

This verifies the permissibility of data streams from the `target` subject to the `source` subject.

It returns the "allowed" result in the following cases:

- The integrity level of the `source` subject does not exceed the integrity level of the `target` subject.
- The integrity level of the `source` subject exceeds the integrity level of the `target` subject, but the minimum integrity level of subjects and resources from which the `source` subject can receive data does not exceed the integrity level of the `target` subject.
- The integrity level of the `source` subject is incomparable to the integrity level of the `target` subject, but the minimum integrity level of subjects and resources from which the `source` subject can receive data does not exceed the integrity level of the `target` subject.

It returns the "denied" result in the following cases:

- The integrity level of the `source` subject exceeds the integrity level of the `target` subject, and the minimum integrity level of subjects and resources from which the `source` subject can receive data exceeds the integrity level of the `target` subject.
- The integrity level of the `source` subject exceeds the integrity level of the `target` subject, and the minimum integrity level of subjects and resources from which the `source` subject can read data is incomparable to the integrity level of the `target` subject.
- The integrity level of the `source` subject is incomparable to the integrity level of the `target` subject, and the minimum integrity level of subjects and resources from which the `source` subject can receive data exceeds the integrity level of the `target` subject.
- The integrity level of the `source` subject is incomparable to the integrity level of the `target` subject, and the minimum integrity level of subjects and resources from which the `source` subject can receive data is incomparable to the integrity level of the `target` subject.
- An integrity level was not assigned to the `source` subject or to the `target` subject.
- The `source` or `target` value is outside of the permissible range.

Example:

```
/* Any client in the solution is allowed to query
 * any server (kernel) if data streams from
 * the server (kernel) to the client are permitted by the
 * Mic security model. Otherwise any client in the solution
 * is prohibited from querying any server (kernel). */
request {
    mic.call { source : src_sid
             , target : dst_sid
             }
}
```

Mic security model invoke rule

```
invoke {source : <Sid>, target : <Sid>}
```

This verifies the permissibility of data streams from the `source` subject to the `target` subject.

It returns the "granted" result if the integrity level of the `target` subject does not exceed the integrity level of the `source` subject.

It returns the "denied" result in the following cases:

- The integrity level of the `target` subject exceeds the integrity level of the `source` subject.
- The integrity level of the `target` subject is incomparable to the integrity level of the `source` subject.
- An integrity level was not assigned to the `source` subject or to the `target` subject.
- The `source` or `target` value is outside of the permissible range.

Mic security model read rule

```
read {source : <Sid>, target : <Sid>}
```

This verifies that the `source` resource consumer is allowed to read data from the `target` resource.

It returns the "allowed" result in the following cases:

- The integrity level of the `source` resource consumer does not exceed the integrity level of the `target` resource.
- The integrity level of the `source` resource consumer exceeds the integrity level of the `target` resource, but the minimum integrity level of subjects and resources from which the `source` resource consumer can receive data does not exceed the integrity level of the `target` resource.
- The integrity level of the `source` resource consumer is incomparable to the integrity level of the `target` resource, but the minimum integrity level of subjects and resources from which the `source` resource consumer can receive data does not exceed the integrity level of the `target` resource.

It returns the "denied" result in the following cases:

- The integrity level of the `source` resource consumer exceeds the integrity level of the `target` resource, and the minimum integrity level of subjects and resources from which the `source` resource consumer can receive data exceeds the integrity level of the `target` resource.
- The integrity level of the `source` resource consumer exceeds the integrity level of the `target` resource, and the minimum integrity level of subjects and resources from which the `source` resource consumer can receive data is incomparable to the integrity level of the `target` resource.
- The integrity level of the `source` resource consumer is incomparable to the integrity level of the `target` resource, and the minimum integrity level of subjects and resources from which the `source` resource consumer can receive data exceeds the integrity level of the `target` resource.
- The integrity level of the `source` resource consumer is incomparable to the integrity level of the `target` resource, and the minimum integrity level of subjects and resources from which the `source` resource consumer can receive data is incomparable to the integrity level of the `target` resource.
- An integrity level was not assigned to the `source` resource consumer or to the `target` resource.
- The `source` or `target` value is outside of the permissible range.

Example:

```
/* Any client in the solution is allowed to query a server of
 * the updater.Realmserv class by calling the read method of the
 * realm.Reader service if the Mic security model permits
 * this client to read data from the resource needed by
 * this client. Otherwise any client in the solution is prohibited from
 * querying a server of the updater.Realmserv class by calling
 * the read method of the realm.Reader endpoint. */
request dst=updater.Realmserv,
        endpoint=realm.Reader {
            match method=read {
                mic.read { source : src_sid
                         , target : message.handle.handle
                         }
```

```
            }
}
```

Mic security model write rule

```
write {source : <Sid>, target : <Sid>}
```

This verifies that the `source` resource consumer is allowed to write data to the `target` resource.

It returns the "granted" result if the integrity level of the `target` resource does not exceed the integrity level of the `source` resource consumer.

It returns the "denied" result in the following cases:

- The integrity level of the `target` resource exceeds the integrity level of the `source` resource consumer.
- The integrity level of the `target` resource is incomparable to the integrity level of the `source` resource consumer.
- An integrity level was not assigned to the `source` resource consumer or to the `target` resource.
- The `source` or `target` value is outside of the permissible range.

Mic security model query_level expression

```
query_level {source : <Sid>}
```

It is intended to be used as an expression that verifies fulfillment of the conditions in the `choice` construct (for details on the `choice` construct, see Binding methods of security models to security events). It checks the integrity level of the `source` resource or subject. Depending on the results of this check, various options for security event handling can be performed.

It runs incorrectly in the following cases:

- An integrity level was not assigned to the subject or `source` resource.
- The `source` value is outside of the permissible range.

When the expression runs incorrectly, the Kaspersky Security Module returns the "denied" decision.

Using templates when creating a KasperskyOS-based solution security policy description

When creating a policy description, you can use the Ginger template engine integrated into the source code generator of the Kaspersky Security Module. To manage the template engine, use the parameters of the CMake command build_vendor_romfs ().

A Ginger PSL template is a PSL file that contains syntactic constructs of the Ginger template engine in addition to declarations in PSL. Files such as `security.psl` and `core.psl` are considered to be Ginger PSL templates if they contain syntactic constructs of the Ginger template engine.

A description of the syntactic constructs of the Ginger template engine is provided in the official documentation for this template engine. For example, you can use macros.

Example of defining a macro:

```
{% macro VMMSafe(src) %}
        match method=Allocate {
            match src={{ src }} { grant () }
        }
{% endmacro %}
```

Example macro call

```
{{ VMMSafe(["Einit"]) }}
```

This call will create the following construct:

```
match method=Allocate {
    match src=Einit { grant () }
}
```

Example definition of a macro with loops:

```
{% macro VMMSafe(srcs) %}
{% for method in ["Allocate", "Commit", "Decommit"] %}
        match method={{ method }} {
{% for src in srcs %}
            match src={{ src }} { grant () }
{% endfor %}
        }
{% endfor %}
{% endmacro %}
```

Example construct with a macro call:

```
match endpoint=vmm.VMM {
{{ VMMSafe(["Einit", "client.Client", "consumer.Consumer",
    "kl.bc.BlobContainer", "kl.VfsEntity", "scm.Producer",
    "execution_manager.ExecMgrEntity", "scm.SecurityConfigMgrEntity",
    "kl.drivers.ATA", "kl.drivers.PCIE"]) }}
}
```

When the macro is expanded in this construct, the following construct will be created:

```
match endpoint=vmm.VMM {
        match method=Allocate {
match src = Einit                               { grant () }
            match src=client.Client { grant () }
            match src=consumer.Consumer { grant () }
            match src=kl.bc.BlobContainer { grant () }
            match src=kl.VfsEntity { grant () }
            match src=scm.Producer { grant () }
            match src=execution_manager.ExecMgrEntity { grant () }
            match src=scm.SecurityConfigMgrEntity { grant () }
            match src=kl.drivers.ATA { grant () }
            match src=kl.drivers.PCIE { grant () }
```

```
        }
        match method=Commit {
match src = Einit                           { grant () }
            match src=client.Client { grant () }
            match src=consumer.Consumer { grant () }
            match src=kl.bc.BlobContainer { grant () }
            match src=kl.VfsEntity { grant () }
            match src=scm.Producer { grant () }
            match src=execution_manager.ExecMgrEntity { grant () }
            match src=scm.SecurityConfigMgrEntity { grant () }
            match src=kl.drivers.ATA { grant () }
            match src=kl.drivers.PCIE { grant () }
        }
        match method=Decommit {
match src = Einit                           { grant () }
            match src=client.Client { grant () }
            match src=consumer.Consumer { grant () }
            match src=kl.bc.BlobContainer { grant () }
            match src=kl.VfsEntity { grant () }
            match src=scm.Producer { grant () }
            match src=execution_manager.ExecMgrEntity { grant () }
            match src=scm.SecurityConfigMgrEntity { grant () }
            match src=kl.drivers.ATA { grant () }
            match src=kl.drivers.PCIE { grant () }
        }
    }
```

Comments can be used in Ginger PSL templates:

```
{# Comment that will be deleted when the template is expanded. #}
```

A file inclusion construct can also be used in Ginger PSL templates.

Example of a file inclusion construct:

```
{% include "defs.ginger" %}
```

The `defs.ginger` file may contain definitions of macros used in a Ginger PSL template, for example. This file must be located in the same directory as the Ginger PSL template that contains this construct.

Methods of KasperskyOS core endpoints

From the perspective of the Kaspersky Security Module, the KasperskyOS kernel is a container of components that provide endpoints. The list of kernel components is provided in the `Core.edl` file located in the `sysroot-*-kos/include/kl/core` directory of the KasperskyOS SDK. This directory also contains the CDL and IDL files for the formal specification of the kernel.

Methods of core endpoints can be divided into secure methods and potentially dangerous methods. Potentially dangerous methods could be used by a cybercriminal in a compromised solution component to cause a denial of service, set up covert data transfer, or hijack an I/O device. Secure methods cannot be used for these purposes.

Access to methods of core endpoints must be restricted as much as possible by the solution security policy (according to the Least Privilege principle). For that, the following requirements must be fulfilled:

1. Access to a secure method must be granted only to the solution components that require this method.
2. Access to a potentially dangerous method must be granted only to the trusted solution components that require this method.

3. Access to a potentially dangerous method must be granted to untrusted solution components that require this method only if the verifiable access conditions limit the possibilities of malicious use of this method, or if the impact from malicious use of this method is acceptable from a security perspective.

   For example, an untrusted component may be allowed to use a limited set of I/O ports that do not allow this component to take control of I/O devices. In another example, covert data transfer between untrusted components may be acceptable from a security perspective.

Virtual memory endpoint

This endpoint is intended for managing virtual memory.

Information about methods of the endpoint is provided in the table below.

Methods of the vmm.VMM endpoint (kl.core.VMM interface)

Method	Method purpose and parameters	Potential danger of the method
`Allocate`	Purpose Allocates (reserves and optionally commits) a virtual memory region. Parameters • [in] `addr` – preferred base address of the virtual memory region, or `0` for the base address to be selected automatically. • [in] `size` – size (in bytes) of the virtual memory region. • [in] `flags` – flags defining the parameters of the virtual memory region. • [out] `va` – base address of the allocated virtual memory region. • [out] `rc` – return code.	Allows the following: • Exhaust the kernel memory by creating a multitude of objects within it. • Exhaust the RAM.
`Commit`	Purpose Commits a virtual memory region that was reserved by the `Allocate` method. Parameters • [in] `va` – base address of the virtual memory region. • [in] `size` – size (in bytes) of the virtual memory region. • [out] `rc` – return code.	Exhausts RAM.

Method	Method purpose and parameters	Potential danger of the method
`Decommit`	Purpose Decommits a virtual memory region. Parameters • [in] `va` – base address of the virtual memory region. • [in] `size` – size (in bytes) of the virtual memory region. • [out] `rc` – return code.	N/A
`Protect`	Purpose Modifies the access rights to the virtual memory region. Parameters • [in] `va` – base address of the virtual memory region. • [in] `size` – size (in bytes) of the virtual memory region. • [in] `flags` – flags defining the access rights to the virtual memory region. • [out] `rc` – return code.	N/A
`Free`	Purpose Frees up the virtual memory region. Parameters • [in] `va` – base address of the virtual memory region. • [in] `size` – size (in bytes) of the virtual memory region. • [out] `rc` – return code.	N/A
`Query`	Purpose Gets information about a virtual memory page. Parameters • [in] `va` – address included in the virtual memory page. • [out] `info` – sequence containing information about a virtual memory page. • [out] `rc` – return code.	N/A
`Reset`	Purpose Decommits a virtual memory region. Parameters • [in] `va` – base address of the virtual memory region. • [in] `size` – size (in bytes) of the virtual memory region. • [out] `rc` – return code.	N/A

Method	Method purpose and parameters	Potential danger of the method
`MdlCreate`	Purpose Creates an MDL buffer. Parameters • [in] `size` – size (in bytes) of the MDL buffer. • [in] `prot` – flags defining the access rights to the MDL buffer. • [out] `handle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the MDL buffer. • [out] `rc` – return code.	Allows the following: • Exhaust the kernel memory by creating a multitude of objects within it. • Exhaust the RAM.
`MdlCreateFromVm`	Purpose Creates an MDL buffer from physical memory that is mapped to the defined virtual memory region and maps the created MDL buffer to this region. Parameters • [in] `va` – base address of the virtual memory region. • [in] `size` – size (in bytes) of the virtual memory region. • [in] `flags` – flags defining the access rights to the MDL buffer. • [out] `handle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the MDL buffer. • [out] `rc` – return code.	Allows the following: • Exhaust the kernel memory by creating a multitude of objects within it. • Exhaust the RAM.
`MdlGetSize`	Purpose Gets the size of the MDL buffer. Parameters • [in] `handle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the MDL buffer. • [out] `size` – size (in bytes) of the MDL buffer. • [out] `rc` – return code.	N/A

Method	Method purpose and parameters	Potential danger of the method
`MdlMap`	Purpose Reserves a virtual memory region and maps the MDL buffer to it. Parameters • [in] `handle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the MDL buffer. • [in] `offset` – offset (in bytes) in the MDL buffer where mapping should start. • [in] `length` – size (in bytes) of the part of the MDL buffer that needs to be mapped. • [in] `hint` – preferred base address of the virtual memory region, or `0` for the base address to be selected automatically. • [in] `prot` – flags defining the parameters of the virtual memory region. • [out] `address` – base address of the virtual memory region. • [out] `rc` – return code.	Allows the following: • Create shared memory for interprocess communication concealed from the security module if multiple processes own the handles of one MDL buffer (the handle permissions masks must allow mapping of the MDL buffer). • Exhaust the kernel memory by creating a multitude of objects within it.
`MdlClone`	Purpose Creates an MDL buffer based on an existing one. The MDL buffer is created from the same regions of physical memory as the original buffer. Parameters • [in] `originHandle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the original MDL buffer. • [in] `offset` – offset (in bytes) in the original MDL buffer where duplication should start. • [in] `length` – size (in bytes) of the part of the original MDL buffer that needs to be duplicated. • [out] `cloneHandle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the created MDL buffer. • [out] `rc` – return code.	Allows the kernel memory to be used up by creating a multitude of objects within it.

Method	Method purpose and parameters	Potential danger of the method
`MdlGetUsageMap`	Purpose Receives information about the arrangement of committed and non-committed pages in a virtual memory region that is mapped to the MDL buffer. Parameters • [in] `handle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the MDL buffer. • [in] `startPage` – virtual memory region offset from which the information collection starts in memory pages. • [out] `firstUsed` – value indicating which counter is the first counter in the `counters` parameter: committed pages counter (non-zero value) or non-committed pages counter (`0`). • [in] `reqCounters` – maximum number of counters in the `counters` parameter. • [out] `counters` – sequence containing counters of committed and non-committed pages. • [out] `rc` – return code.	N/A

Physical memory endpoint

This endpoint is intended for performing operations with a working set.

Information about methods of the endpoint is provided in the table below.

Methods of the pmm.PMM endpoint (kl.core.PMM interface)

Method	Method purpose and parameters	Potential danger of the method
`WsOpen`	Purpose Provides access to the working set. Parameters • [out] `ws` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the working set. • [out] `rc` – return code.	N/A

Method	Method purpose and parameters	Potential danger of the method
`WsGetInfo`	Purpose Gets the characteristics of the working set. Parameters • [in] `ws` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the working set. • [out] `info` – the structure containing the characteristics of the working set. • [out] `rc` – return code.	N/A
`WsSetThreshold`	Purpose Sets the threshold values for the amount of used physical memory in the system. Parameters • [in] `ws` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the working set. • [in] `threshold` – the lower threshold value, in memory pages. • [in] `width` – the offset of the upper threshold value relative to the lower one, in memory pages. • [out] `rc` – return code.	Allows the following: • Change the threshold values set by another process (if the `WsClearThreshold` method is available). • Set the threshold values after deletion by another process.
`WsClearThreshold`	Purpose Delete the threshold values for the amount of used physical memory in the system. Parameters • [in] `ws` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the working set. • [out] `rc` – return code.	Removes threshold values that were set by another process.

I/O endpoint

This endpoint is intended for working with I/O ports, MMIO, DMA, and interrupts.

The `EnumEnrichchmentIds`, `EnrichResourceGroup`, and `RevokeEnrichmentResource` methods are intended to be used by PinCtrl class drivers. Information about all other methods of the endpoint is provided in the table below.

Methods of the io.IO endpoint (kl.core.IO interface)

Method	Method purpose and parameters	Potential danger of the method
`RegisterPort`	Purpose Registers a sequence of I/O ports. Parameters • [in] `base` – address of the first I/O port in the sequence. • [in] `size` – number of I/O ports in the sequence. • [out] `resource` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the sequence of I/O ports. • [out] `rc` – return code.	Allows the following: • Hijack I/O ports (it is recommended to monitor the address of the first I/O port and the number of I/O ports in the sequence). • Exhaust the kernel memory by creating a multitude of objects within it.
`RegisterMmio`	Purpose Registers an MMIO memory region. Parameters • [in] `base` – base address of the MMIO memory region. • [in] `size` – size (in bytes) of the MMIO memory region. • [in] `flags` – flags that specify the parameters of the MMIO memory region. • [out] `resource` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the MMIO memory region. • [out] `rc` – return code.	Allows the kernel memory to be used up by creating a multitude of objects within it.
`RegisterDma`	Purpose Creates a DMA buffer within the specified region of physical memory. Parameters • [in] `from` – base address of the physical memory region. • [in] `to` – last address of the physical memory region. • [in] `size` – size (in bytes) of the DMA buffer. • [in] `flags` – flags defining the DMA buffer parameters. • [in] `order` – parameter defining the minimum number of memory pages (*2^order*) in a block. • [out] `resource` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the DMA buffer. • [out] `rc` – return code.	Allows the following: • Exhaust the kernel memory by creating a multitude of objects within it. • Exhaust the RAM.

Method	Method purpose and parameters	Potential danger of the method
`RegisterIrq`	Purpose Registers an interrupt that occurs when a device queries an interrupt controller through a hardware interrupt line. Parameters • [in] `irq` – interrupt number. • [in] `flags` – flags defining the interrupt parameters. • [out] `resource` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the interrupt. • [out] `rc` – return code.	Allows the kernel memory to be used up by creating a multitude of objects within it.
`MapMem`	Purpose Reserves the virtual memory region and maps the MMIO memory region to it. Parameters • [in] `resource` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the MMIO memory region. • [in] `prot` – flags defining the access rights to the virtual memory region. • [in] `attr` – flags defining the parameters of the virtual memory region (for example, use of caching). • [out] `address` – base address of the virtual memory region. • [out] `mapping` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle is used to free the virtual memory region. • [out] `rc` – return code.	Allows the following: • Take control of a device when mapping an MMIO memory region to a virtual memory region (it is recommended to monitor the base address and size of the MMIO memory region when the `RegisterMmio` method is called). • Create shared memory for interprocess communication concealed from the security module if multiple processes own the handles of one MMIO memory region (the handle permissions masks must allow mapping of the MMIO memory region). • Exhaust the kernel memory by creating a multitude of objects within it.
`PermitPort`	Purpose Opens access to I/O ports. Parameters • [in] `resource` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the sequence of I/O ports. • [out] `access` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle is used to close access to I/O ports. • [out] `rc` – return code.	Allows the following: • Take control of a device (it is recommended to monitor the address of the first I/O port and the number of I/O ports in the sequence when the `RegisterPort` method is called). • Exhaust the kernel memory by creating a multitude of objects within it.

Method	Method purpose and parameters	Potential danger of the method
`AttachIrq`	Purpose Attaches the calling thread to an interrupt. Parameters • [in] `resource` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the interrupt. • [out] `delivery` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle is the client IPC handle that is used by the interrupt handler. • [out] `rc` – return code.	Allows the following: • Take CPU time from all other threads, including from other processes (the thread that attached to the interrupt will become a real-time thread). • Make it impossible to terminate a process from another process (the process whose thread was attached to the interrupt cannot be terminated from another process). • Stop the operating system (if an unhandled exception occurs in the thread handling an interrupt, the operating system stops). • Lock, delay, or incorrectly handle an interrupt (it is recommended to monitor the interrupt number when the `RegisterIrq` method is called). • Exhaust the kernel memory by creating a multitude of objects within it.
`DetachIrq`	Purpose Sends a request to a thread. When this request is fulfilled, the thread must detach from the interrupt. Parameters • [in] `resource` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the interrupt. • [out] `rc` – return code.	Stops interrupt handling in another process.
`EnableIrq`	Purpose Enables (unmasks) an interrupt. Parameters • [in] `resource` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the interrupt. • [out] `rc` – return code.	Enables an interrupt at the system level.
`DisableIrq`	Purpose Disables (masks) an interrupt. Parameters • [in] `resource` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the interrupt. • [in] `sync` – value that defines whether to wait for completion of the current iteration in the maskable interrupt handler (a non-zero value means yes, while `0` means no). • [out] `rc` – return code.	Disables an interrupt at the system level.

Method	Method purpose and parameters	Potential danger of the method
`ModifyDma`	Purpose Modifies the DMA buffer cache settings. Parameters • [in] `resource` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the DMA buffer. • [in] `flags` – flags defining the DMA buffer caching parameters. • [out] `rc` – return code.	N/A
`MapDma`	Purpose Reserves a virtual memory region and maps the DMA buffer to it. Parameters • [in] `resource` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the DMA buffer. • [in] `offset` – offset (in bytes) in the DMA buffer where mapping should start. • [in] `length` – size (in bytes) of the part of the DMA buffer that needs to be mapped. • [in] `hint` – preferred base address of the virtual memory region, or `0` for the base address to be selected automatically. • [in] `prot` – flags defining the access rights to the virtual memory region. • [out] `address` – base address of the virtual memory region. • [out] `mapping` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle is used to free the virtual memory region. • [out] `rc` – return code.	Allows the following: • Create shared memory for interprocess communication concealed from the security module if multiple processes own the handles of one DMA buffer (the handle permissions masks must allow mapping of the DMA buffer). • Exhaust the kernel memory by creating a multitude of objects within it.
`DmaGetInfo`	Purpose Gets information about a DMA buffer. Parameters • [in] `resource` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the DMA buffer. • [out] `flags` – flags indicating the DMA parameters. • [out] `order` – parameter indicating the minimum number of memory pages (2^order) in a block. • [out] `size` – size (in bytes) of the DMA buffer. • [out] `count` – number of blocks. • [out] `frames` – sequence containing the addresses and sizes of blocks. • [out] `rc` – return code.	N/A

Method	Method purpose and parameters	Potential danger of the method
`DmaGetPhysInfo`	Purpose Gets the physical addresses and sizes of DMA buffer blocks. Parameters • [in] `handle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the DMA buffer. • [out] `count` – number of blocks. • [out] `frames` – sequence containing the physical addresses and sizes of blocks. • [out] `rc` – return code.	N/A
`BeginDma`	Purpose Opens access to a DMA buffer for a device. Parameters • [in] `resource` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the DMA buffer. • [in] `domain` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the IOMMU domain. • [out] `iomapping` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the kernel object that contains the addresses and sizes of blocks required by the device to use the DMA buffer. The memory addresses used by the device can be physical addresses or virtual addresses depending on whether the IOMMU is enabled. • [out] `rc` – return code.	Allows the kernel memory to be used up by creating a multitude of objects within it.
`AllocMsi`	Purpose Creates a message signaled interrupt object. Parameters • [in] `order` – fictitious parameter. It must be set to `0`. • [in] `deviceId` – address of the device on the PCIe bus in BDF format. • [in] `priority` – interrupt priority. • [out] `addr` – MMIO memory address to which the device needs to write to trigger an interrupt (Message Address according to the PCIe specification). • [out] `data` – data that the device needs to write to the MMIO memory at the address obtained through the `addr` parameter to trigger an interrupt (Message Data according to the PCIe specification). • [out] `msi` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the message signaled interrupt object. • [out] `rc` – return code.	Allows the following: • Exhaust the available interrupt vectors. • Exhaust the kernel memory by creating a multitude of objects within it.

Method	Method purpose and parameters	Potential danger of the method
`RegisterMsiIrq`	Purpose Registers a message signaled interrupt. Parameters • [in] `msi` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the message signaled interrupt object. • [in] `index` – fictitious parameter. It must be set to `0`. • [out] `irq` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the interrupt. • [out] `rc` – return code.	Allows the kernel memory to be used up by creating a multitude of objects within it.

Threads endpoint

This endpoint is intended for managing threads.

Information about methods of the endpoint is provided in the table below.

Methods of the thread.Thread endpoint (kl.core.Thread interface)

Method	Method purpose and parameters	Potential danger of the method
`Create`	Purpose Creates a thread. Parameters • [out] `thread` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the thread. • [out] `tid` – thread ID (TID). • [in] `priority` – thread priority. • [in] `stackSize` – thread stack limit (in bytes), or `0` to use the default size that was defined when the process was created. • [in] `routine` – pointer to the function that is called when the thread starts. • [in] `context` – pointer to the function executed by the thread. • [in] `context2` – pointer to the parameters passed to the function defined via the `context` parameter. • [in] `tls` – pointer to the TLS. • [in] `flags` – flags defining the parameters for creating the thread. • [out] `rc` – return code.	Allows the following: • Create a real-time thread that takes up all the CPU time from other threads, including from other processes (it is recommended to monitor thread creation parameters). • Create a multitude of threads (including with high priority) to reduce the CPU time available to the threads of other processes (it is recommended to monitor thread priority). • Exhaust the RAM. • Exhaust the kernel memory by creating a multitude of objects within it.

Method	Method purpose and parameters	Potential danger of the method
`OpenCurrent`	Purpose Creates the handle of the calling thread. Parameters • [out] `thread` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the thread. • [out] `rc` – return code.	N/A
`Suspend`	Purpose Locks the calling thread. Parameters • [out] `rc` – return code.	Locks a thread that has captured a synchronization object that was created in shared memory and is anticipated by a thread of another process. As a result, the thread of the other process may be locked indefinitely.
`Resume`	Purpose Resumes execution of a locked thread. Parameters • [in] `thread` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the thread. • [out] `rc` – return code.	N/A
`Terminate`	Purpose Terminates the locked thread. Parameters • [in] `thread` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the thread. • [in] `code` – thread exit code. • [out] `tid` – thread ID (TID). • [out] `rc` – return code.	N/A
`Exit`	Purpose Terminates the calling thread. Parameters • [in] `code` – thread exit code. • [out] `rc` – return code.	N/A

Method	Method purpose and parameters	Potential danger of the method
`Wait`	Purpose Locks the calling thread until the defined thread is terminated. Parameters • [in] `thread` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the thread. • [in] `msec` – thread termination timeout (in milliseconds). • [out] `code` – thread exit code. • [out] `rc` – return code.	N/A
`SetPriority`	Purpose Defines the priority of a thread. Parameters • [in] `thread` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the thread. • [in] `priority` – thread priority. • [out] `rc` – return code.	Allows the priority of a thread to be elevated to reduce the CPU time available to all other threads, including from other processes. It is recommended to monitor thread priority.
`SetTls`	Purpose Sets the base address of the TLS for the calling thread. Parameters • [in] `va` – pointer to the TLS. • [out] `rc` – return code.	N/A
`Sleep`	Purpose Locks the calling thread for the specified duration. Parameters • [in] `mdelay` – thread blocking duration (in milliseconds). • [out] `rc` – return code.	N/A

Method	Method purpose and parameters	Potential danger of the method
`GetInfo`	Purpose Gets information about a thread. Parameters • [in] `thread` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the thread. • [out] `info` – structure containing the base address of the thread stack and its size (in bytes), and the thread identifier (TID). • [out] `rc` – return code.	N/A
`DetachIrq`	Purpose Detaches the calling thread from the interrupt handled in its context. Parameters • [out] `rc` – return code.	N/A
`GetAffinity`	Purpose Gets a thread affinity mask. Parameters • [in] `thread` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the thread. • [out] `mask` – thread affinity mask. • [out] `rc` – return code.	N/A
`SetAffinity`	Purpose Sets a thread affinity mask. Parameters • [in] `thread` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the thread. • [in] `mask` – thread affinity mask. • [out] `rc` – return code.	N/A

Method	Method purpose and parameters	Potential danger of the method
`SetSchedPolicy`	Purpose Defines the scheduler class and priority of the thread. Parameters • [in] `thread` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the thread. • [in] `policy` – thread scheduler class. • [in] `priority` – thread priority. • [in] `param` – union containing parameters of a thread scheduler class. • [out] `rc` – return code.	Allows the following: • Convert a thread into a real-time thread that takes up all the CPU time from all other threads, including from other processes (it is recommended to monitor the thread scheduler class). • Elevate the priority of a thread to reduce the CPU time available to all other threads, including from other processes (it is recommended to monitor thread priority).
`GetSchedPolicy`	Purpose Gets information about the scheduler class and priority of a thread. Parameters • [in] `thread` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the thread. • [out] `policy` – thread scheduler class. • [out] `priority` – thread priority. • [out] `param` – union containing parameters of a thread scheduler class. • [out] `rc` – return code.	N/A

Handles endpoint

This endpoint is intended for performing operations with handles.

Information about methods of the endpoint is provided in the table below.

Methods of the handle.Handle endpoint (kl.core.Handle interface)

Method	Method purpose and parameters	Potential danger of the method
`Copy`	Purpose Duplicates a handle. As a result of duplication, the calling process receives the handle descendant. Parameters • [in] `inHandle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle field contains the original handle. • [in] `newRightsMask` – permissions mask of the handle descendant. • [in] `copyBadge` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the resource transfer context object. • [out] `outHandle` – value whose binary representation consists of multiple fields, including a field for the handle descendant and a field for the permissions mask of the handle descendant. • [out] `rc` – return code.	Allows the kernel memory to be used up by creating a multitude of objects within it.
`CreateUserObject`	Purpose Creates a handle. Parameters • [in] `type` – handle type. • [in] `rights` – handle permissions mask. • [in] `context` – pointer to the data that should be associated with the handle. • [in] `ipcChannel` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle is the server IPC handle. • [in] `riid` – Runtime Implementation Identifier (RIID). • [out] `handle` – value whose binary representation consists of multiple fields, including a field for the created handle and a field for the permissions mask of the created handle. • [out] `rc` – return code.	Allows the kernel memory to be used up by creating a multitude of objects within it.
`Close`	Purpose Closes a handle. Parameters • [in] `handle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. • [out] `rc` – return code.	N/A

Method	Method purpose and parameters	Potential danger of the method
`Connect`	Purpose Creates and connects the client, server, and listener IPC handles. Parameters • [in] `server` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the server process. • [in] `srListener` – listener handle from the handle space of the server process, or the value `0xFFFFFFFF` to create it. • [in] `createSrEndpoint` – value that defines whether or not to create a server IPC handle in the handle space of the server process (non-zero value for yes, `0` for no). • [in] `client` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the client process. • [out] `outSrListener` – listener handle from the handle space of the server process. • [out] `outSrEndpoint` – server IPC handle from the handle space of the server process. • [out] `outClEndpoint` – client IPC handle from the handle space of the client process. • [out] `rc` – return code.	Allows the kernel memory to be used up by creating a multitude of objects within it.
`SecurityConnect`	Purpose Creates a client IPC handle for querying the Kaspersky Security Module through the security interface. Parameters • [out] `client` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. • [out] `rc` – return code.	Allows a multitude of possible kernel process handle values to be used up.
`GetSidByHandle`	Purpose Receives a security ID (SID) based on a handle. Parameters • [in] `handle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. • [out] `sid` – security ID (SID). • [out] `rc` – return code.	N/A
`Revoke`	Purpose Revokes the descendants of the specified handle and (optionally) closes the specified handle. Parameters • [in] `handle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. • [in] `close` – value that determines whether to close the handle specified in the `handle` parameter (non-zero value – yes, `0` – no). • [out] `rc` – return code.	N/A

Method	Method purpose and parameters	Potential danger of the method
`RevokeSubtree`	Purpose Revokes the handles that make up the inheritance subtree of the specified handle. Parameters • [in] `handle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handles forming the inheritance subtree of this handle are revoked. • [in] `badge` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the resource transfer context object that defines the inheritance subtree of the handles to revoke. The root node of this subtree is the handle that was generated by the transfer or duplication of the handle that is defined through the `handle` parameter and is associated with the resource transfer context object. • [out] `rc` – return code.	N/A
`CreateBadge`	Purpose Creates a resource transfer context object and configures a notification mechanism for monitoring the life cycle of this object. Parameters • [in] `notify` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the notification receiver. • [in] `notifyContext` – ID of the "resource–event mask" entry in the notification receiver. • [in] `badgeContext` – pointer to the data that should be associated with the handle transfer. • [out] `badge` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the resource transfer context object. • [out] `rc` – return code.	Allows the kernel memory to be used up by creating a multitude of objects within it.

Processes endpoint

This endpoint is intended for managing processes.

Information about methods of the endpoint is provided in the table below.

Methods of the task.Task endpoint (kl.core.Task interface)

Method	Method purpose and parameters	Potential danger of the method
`Create`	Purpose Creates a process. Parameters • [in] `name` – process name. • [in] `eiid` – process class name. • [in] `path` – path to the executable file. • [in] `stackSize` – thread stack limit (in bytes) used by default when creating process threads. • [in] `priority` – priority of the initial thread. • [in] `flags` – flags defining the parameters for creating the process. • [out] `child` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the created process. • [out] `rc` – return code.	Allows the following: • Create a process that will be privileged from the perspective of the solution security policy (indicating the name of the process class with privileges). • Reserve a process name so that another process with this name cannot be created. • Create a process that will cause the operating system to stop if an unhandled exception occurs. • Load code from an executable file into process memory for subsequent execution of that code. • Exhaust RAM by creating a multitude of processes. • Exhaust the kernel memory by creating a multitude of objects within it.
`LoadSeg`	Purpose Loads an ELF image segment into process memory. Parameters • [in] `task` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the process. • [in] `mdl` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the MDL buffer containing the loaded ELF image segment. • [in] `segAttr` – structure containing the parameters for loading the ELF image segment. • [out] `rc` – return code. • [out] `retaddr` – base address of the virtual memory region of the process where the ELF image segment is loaded.	Allows code to be loaded into process memory for subsequent execution of that code.

Method	Method purpose and parameters	Potential danger of the method
`VmReserve`	Purpose Reserves the virtual memory region in a process that was created as an empty process. Parameters • [in] `task` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the process. • [in] `addr` – preferred base address of the virtual memory region, or `0` for the address to be selected automatically. • [in] `size` – size (in bytes) of the virtual memory region. • [in] `flags` – flags defining the parameters of the virtual memory region. • [out] `outAddr` – base address of the reserved virtual memory region. • [out] `rc` – return code.	Allows the following: • Exhaust the kernel memory by creating a multitude of objects within it. • Reserve virtual memory regions in another process that was created as an empty process and has not yet been started (if its handle is available). The handle permissions mask must allow reservation of virtual memory.
`VmFree`	Purpose Frees the virtual memory region that was reserved by calling the `VmReserve` method in a process that was created as an empty process. Parameters • [in] `task` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the process. • [in] `addr` – base address of the virtual memory region. • [in] `size` – size (in bytes) of the virtual memory region. • [out] `rc` – return code.	Frees virtual memory regions in another process that was created as an empty process and has not yet been started (if its handle is available). (The handle permissions mask must allow freeing of virtual memory.)
`SetEntry`	Purpose Defines the program entry point and the ELF image load offset. Parameters • [in] `task` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the process. • [in] `state` – structure containing the address of the program entry point and the ELF image load offset (in bytes). • [out] `rc` – return code.	Creates conditions for executing code loaded into process memory.

Method	Method purpose and parameters	Potential danger of the method
`LoadElfSyms`	Purpose Loads the symbol table `.symtab` and string table `.strtab` into the memory of a process that was created as an empty process. Parameters • [in] `task` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the process. • [in] `symMdl` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the MDL buffer containing the symbol table `.symtab`. • [in] `symSegAttr` – structure containing the parameters for loading the symbol table `.symtab`. • [in] `symSize` – size of the symbol table `.symtab` (in bytes). • [in] `strMdl` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the MDL buffer containing the string table `.strtab`. • [in] `strSegAttr` – structure containing the parameters for loading the string table `.strtab`. • [in] `strSize` – size of the string table `.strtab` (in bytes). • [out] `rc` – return code.	N/A
`SetEnv`	Purpose Writes data to the SCP of a child process. Parameters • [in] `task` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the child process. • [in] `env` – sequence containing data to be written to the SCP. • [out] `rc` – return code.	Allows the kernel memory to be used up by creating a multitude of objects within it.
`FreeSelfEnv`	Purpose Deletes the SCP of the calling process. Parameters • [out] `rc` – return code.	N/A

Method	Method purpose and parameters	Potential danger of the method
`Resume`	Purpose Resumes a process. Parameters • [in] `task` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the process. • [out] `rc` – return code.	Allows the following: • Execute code loaded into process memory. • Start a multitude of previously created processes to reduce the computing resources available to other processes (it is recommended to monitor the priority of the initial thread when the `Create` method is called).
`Exit`	Purpose Terminates the calling process. Parameters • [in] `status` – exit code of the process. • [out] `rc` – return code.	N/A
`Terminate`	Purpose Terminates a process. Parameters • [in] `task` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the process. • [out] `rc` – return code.	Allows another process to be terminated if its handle is available. (The handle permissions mask must allow termination of the process.)
`GetExitInfo`	Purpose Gets information about a terminated process. Parameters • [in] `task` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the terminated process. • [out] `status` – value indicating the reason for process termination. • [out] `info` – union containing information about the terminated process. • [out] `rc` – return code.	N/A

Method	Method purpose and parameters	Potential danger of the method
`GetThreadContext`	Purpose Gets the context of a thread that is part of a frozen process. Parameters • [in] `task` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the process that is in a frozen state. • [in] `index` – thread index. It is used to enumerate threads. Enumeration starts with zero. A thread in which an unhandled exception occurred has a zero index. • [out] `context` – structure containing the thread ID (TID) and thread context. • [out] `rc` – return code.	Enables disrupted isolation of a process that is in a frozen state. For example, the thread context may contain the values of variables.
`GetNextVmRegion`	Purpose Gets information about the virtual memory region that belongs to a frozen process. Parameters • [in] `task` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the process that is in a frozen state. • [in] `after` – address that is followed by the virtual memory region. • [in] `needMdl` – value that indicates whether or not you need to create an MDL buffer from physical memory that is mapped to a virtual memory region (a non-zero value means yes and `0` means no). • [out] `next` – base address of the virtual memory region. • [out] `size` – size of the virtual memory region (in bytes). • [out] `flags` – flags indicating the parameters of the virtual memory region. • [out] `handle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the MDL buffer mapped to a virtual memory region. • [out] `rc` – return code.	Enables disrupted isolation of a process that is in a frozen state. Process isolation is disrupted due to the opened access to the process memory region.

Method	Method purpose and parameters	Potential danger of the method
`TerminateAfterFreezing`	Purpose Terminates a frozen process. Parameters • [in] `task` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the process that is in a frozen state. • [out] `rc` – return code.	Enables termination of a frozen process. This does not allow collection of data about this process for diagnostic purposes.
`GetName`	Purpose Gets the name of a calling process. Parameters • [out] `name` – process name. • [out] `rc` – return code.	N/A
`GetPath`	Purpose Gets the path to the executable file from which the calling process was created. Parameters • [out] `path` – path to the executable file. • [out] `rc` – return code.	N/A
`GetInitialThreadPriority`	Purpose Gets the priority of the initial thread of a process. Parameters • [in] `task` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the process. • [out] `priority` – priority of the initial thread. • [out] `rc` – return code.	N/A
`SetInitialThreadPriority`	Purpose Sets the priority of the initial thread of a process. Parameters • [in] `task` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the process. • [in] `priority` – priority of the initial thread. • [out] `rc` – return code.	Allows the priority of the initial thread of a process to be elevated to reduce the CPU time available to all other threads, including from other processes. It is recommended to monitor the priority of an initial thread.

Method	Method purpose and parameters	Potential danger of the method
`GetTasksList`	Purpose Gets information about existing processes. Parameters • [out] `notice` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the notification receiver that is configured to receive notifications regarding the termination of processes. • [out] `strings` – sequence containing the parameters of processes. • [out] `pids` – sequence containing the identifiers of processes (the PID of each process). • [out] `rc` – return code.	Allows the kernel memory to be used up by creating a multitude of objects within it.
`SetInitialThreadSchedPolicy`	Purpose Defines the scheduler class and priority of the initial thread of a process. Parameters • [in] `task` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the process. • [in] `policy` – scheduler class of the initial thread of the process. • [in] `priority` – priority of the initial thread of a process. • [in] `params` – union containing the parameters of the scheduler class of the initial thread of the process. • [out] `rc` – return code.	Allows the following: • Convert the initial thread of a process into a real-time thread that takes up all the CPU time from all other threads, including from other processes (it is recommended to monitor the scheduler class of the initial thread of the process). • Elevate the priority of the initial thread of a process to reduce the CPU time available to all other threads, including from other processes (it is recommended to monitor the priority of the initial thread of the process).
`ReseedAslr`	Purpose Defines the seed value for ASLR support. Parameters • [in] `task` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the process. • [in] `seed` – sequence containing the seed value. • [out] `rc` – return code.	N/A

Method	Method purpose and parameters	Potential danger of the method
`GetElfSyms`	Purpose Gets the address and size of the symbol table `.symtab` and string table `.strtab` for the calling process. Parameters • [out] `relocBase` – ELF image load offset (in bytes). • [out] `syms` – address of the symbol table `.symtab`. • [out] `symsCnt` – size (in bytes) of the symbol table `.symtab`. • [out] `strs` – address of the string table `.strtab`. • [out] `strsSize` – size (in bytes) of the string table `.strtab`. • [out] `rc` – return code.	N/A
`TransferHandle`	Purpose Transfers a handle to a process that is not yet running. Parameters • [in] `task` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the process. • [in] `srcHandle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle field contains the transferred handle. • [in] `srcBadge` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the resource transfer context object. • [in] `dstRights` – permissions mask of the descendant of the transferred handle. • [out] `dstHandle` – value of the descendant of the transferred handle (from the handle space of the process that received the handle). • [out] `rc` – return code.	Allows the kernel memory to be used up by creating a multitude of objects within it.
`GetPid`	Purpose Gets the process ID (PID). Parameters • [in] `task` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the process. • [out] `pid` – process ID. • [out] `rc` – return code.	N/A

Method	Method purpose and parameters	Potential danger of the method
`SetEntryExceptionHandler`	Purpose Defines the entry point to the exception handler for the calling process. Parameters • [in] `entry` – address of the entry point to the exception handler that will be used. • [out] `prev` – address of the entry point to the previous exception handler. • [out] `rc` – return code.	N/A

Synchronization endpoint

The endpoint facilitates using futexes to synchronize threads.

Information about methods of the endpoint is provided in the table below.

Methods of the sync.Sync endpoint (kl.core.Sync interface)

Method	Method purpose and parameters	Potential danger of the method
`Wait`	Purpose Locks execution of the calling thread if the futex value is equal to the expected value. Futex is linked to a queue of locked threads arranged according to the FIFO principle. Parameters • [in] `ptr` – pointer to the variable containing the futex value. • [in] `val` – expected value of the futex. • [in] `delay` – maximum blocking duration (in milliseconds). • [out] `outDelay` – actual blocking duration (in milliseconds). • [out] `rc` – return code.	N/A
`Wake`	Purpose Resumes execution of threads that are the first in the queue of threads that were locked by calls of the `Wait` method with the specified futex. Futex is linked to a queue of locked threads arranged according to the FIFO principle. Parameters • [in] `ptr` – pointer to the variable containing the futex value. • [in] `nThreads` – maximum number of threads whose execution will be resumed. • [out] `wokenCnt` – actual number of threads whose execution was resumed. • [out] `rc` – return code.	N/A

Method	Method purpose and parameters	Potential danger of the method
`PiWait`	Purpose Locks the calling thread. Futex is linked to a queue of locked threads arranged in the order of their priority. Parameters • [in] `ptr` – pointer to the variable containing the futex value. • [in] `delay` – maximum blocking duration (in milliseconds). • [out] `rc` – return code.	N/A
`PiWake`	Purpose Resumes execution of a thread that is the first in the queue of threads that were locked by calls of the `PiWait` method with the specified futex. Futex is linked to a queue of locked threads arranged in the order of their priority. Parameters • [in] `ptr` – pointer to the variable containing the futex value. • [out] `rc` – return code.	N/A

File system endpoint

This endpoint is intended for working with the ROMFS file system managed by the KasperskyOS kernel.

Information about methods of the endpoint is provided in the table below.

Methods of the fs.FS endpoint (kl.core.FS interface)

Method	Method purpose and parameters	Potential danger of the method
`Open`	Purpose Opens a file. Parameters • [in] `name` – name of the file. • [out] `handle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the file. • [out] `rc` – return code.	Allows the kernel memory to be used up by creating a multitude of objects within it.
`Close`	Purpose Closes a file. Parameters • [in] `handle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the file. • [out] `rc` – return code.	N/A

Method	Method purpose and parameters	Potential danger of the method
`Read`	Purpose Reads data from a file. Parameters • [in] `handle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the file. • [in] `sectorNumber` – data block number. Enumeration starts with zero. • [out] `read` – size (in bytes) of the read data. • [out] `data` – sequence containing the read data. • [out] `rc` – return code.	N/A
`GetSize`	Purpose Gets the size of a file. Parameters • [in] `handle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the file. • [out] `size` – file size (in bytes). • [out] `rc` – return code.	N/A
`GetId`	Purpose Gets the unique ID of a file. Parameters • [in] `handle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the file. • [out] `id` – unique ID of the file. • [out] `rc` – return code.	N/A
`Count`	Purpose Gets the number of files in the file system. Parameters • [out] `count` – number of files in the file system. • [out] `rc` – return code.	N/A

Method	Method purpose and parameters	Potential danger of the method
`GetInfo`	Purpose Gets the name and unique ID of a file based on the file index. Parameters • [in] `index` – file index. Enumeration starts with zero. • [in] `nameLenMax` – buffer size (in bytes) for saving the file name. • [out] `name` – name of the file. • [out] `id` – unique ID of the file. • [out] `rc` – return code.	N/A
`GetFsSize`	Purpose Gets the size of the file system. Parameters • [out] `fsSize` – size (in bytes) of the file system. • [out] `rc` – return code.	N/A

Time endpoint

This endpoint is designed to set the system time and get RTC characteristics.

Information about methods of the endpoint is provided in the table below.

Methods of the time.Time endpoint (kl.core.Time interface)

Method	Method purpose and parameters	Potential danger of the method
`SetSystemTime`	Purpose Sets the system time. Parameters • [in] `secs` – time (in seconds) that has elapsed since January 1, 1970. • [in] `nsecs` – additional time (in nanoseconds) added to the time defined through the `secs` parameter. • [out] `rc` – return code.	Allows the system time to be set.

Method	Method purpose and parameters	Potential danger of the method
`SetSystemTimeAdj`	Purpose Starts gradual adjustment of the system time. Parameters • [in] `adj` – structure containing the amount of time by which the system time must be adjusted (*sec*10^9+nsec* nanoseconds). • [in] `slew` – rate of system time adjustment (microseconds per second). • [out] `prev` – structure containing the correction time value that remained for system time adjustment to be completed for the previous gradual time adjustment (*sec*10^9+nsec* nanoseconds). • [out] `rc` – return code.	Allows the system time to be changed.
`GetSystemTimeAdj`	Purpose Gets the correction time value remaining for system time adjustment so that gradual adjustment can be fully completed. Parameters • [out] `adj` – structure containing the correction time value remaining for system time adjustment so that the gradual adjustment can be fully completed (*sec*10^9+nsec* nanoseconds). • [out] `rc` – return code.	N/A
`GetRtcInfo`	Purpose Gets RTC characteristics. Parameters • [out] `info` – structure containing RTC characteristics: resolution and the lower and upper bounds of the countdown time (relative to January 1, 1970). • [out] `rc` – return code.	N/A

Hardware abstraction layer endpoint

This endpoint is designed to obtain the values of hardware platform parameters, read and set privileged registers of peripheral devices, flush the processor cache, forward diagnostic data to the logging system, and receive arrays of hardware-generated random byte values.

Information about methods of the endpoint is provided in the table below.

Methods of the hal.HAL endpoint (kl.core.HAL interface)

Method	Method purpose and parameters	Potential danger of the method
`GetEnv`	Purpose Gets the value of a hardware platform parameter. Parameters • [in] `name` – name of the parameter. • [out] `value` – value of the parameter. • [out] `rc` – return code.	Gets values of hardware platform parameters that could contain critical system information.
`GetPrivReg`	Purpose Gets the value of a peripheral device privileged register. Parameters • [in] `reg` – name of the register. • [out] `val` – value of the register. • [out] `rc` – return code.	Sets up a data transfer channel with a process that has access to the `SetPrivReg` or `SetPrivRegRange` method. It is recommended to monitor the name of a register.
`SetPrivReg`	Purpose Sets the value of a peripheral device privileged register. Parameters • [in] `reg` – name of the register. • [in] `val` – value of the register. • [out] `rc` – return code.	Allows the following: • Set the value of a peripheral device privileged register. • Set up a data transfer channel with a process that has access to the `GetPrivReg` or `GetPrivRegRange` method. It is recommended to monitor the name of a register.
`GetPrivRegRange`	Purpose Gets the value of a peripheral device privileged register. Parameters • [in] `regRange` – name of the registers range. • [in] `offset` – register offset in the registers range. • [out] `val` – value of the register. • [out] `rc` – return code.	Sets up a data transfer channel with a process that has access to the `SetPrivReg` or `SetPrivRegRange` method. It is recommended to monitor the name of the registers range and the register offset in this range.

Method	Method purpose and parameters	Potential danger of the method
`SetPrivRegRange`	Purpose Sets the value of a peripheral device privileged register. Parameters • [in] `regRange` – name of the registers range. • [in] `offset` – register offset in the registers range. • [in] `val` – value of the register. • [out] `rc` – return code.	Allows the following: • Set the value of a peripheral device privileged register. • Set up a data transfer channel with a process that has access to the `GetPrivReg` or `GetPrivRegRange` method. It is recommended to monitor the name of the registers range and the register offset in this range.
`FlushCache`	Purpose Flushes the processor cache. Parameters • [in] `type` – value defining the cache type (data cache, instructions cache, or joint data and instructions cache). • [in] `va` – base address of the virtual memory region. The cache corresponding to this region is flushed. • [in] `size` – size (in bytes) of the virtual memory region. The cache corresponding to this region is flushed. • [out] `rc` – return code.	Allows the processor cache to be flushed.
`DebugWrite`	Purpose Forwards diagnostic data to the logging system. Parameters • [in] `data` – sequence containing the diagnostic data. • [out] `rc` – return code.	Passes fictitious data (for example, uninformative data) to the logging system.
`GetEntropy`	Purpose Gets an array of hardware-generated random byte values. Parameters • [out] `buffer` – sequence containing random byte values. • [in] `size` – number of random byte values. • [out] `rc` – return code.	Creates a load on the hardware-based random number generator with frequent method calls to restrict the use of this random number generator for other processes.

Method	Method purpose and parameters	Potential danger of the method
`GetEntropyInfo`	Purpose Gets information about the source of entropy. Parameters • [out] `info` – structure containing the min-entropy value. • [out] `rc` – return code.	N/A

KasperskyOS kernel XHCI DbC driver management endpoint

This endpoint is intended for use by drivers that implement a USB stack (kusb class drivers) and lets you temporarily suspend the XHCI DbC driver of the KasperskyOS kernel.

Information about methods of the endpoint is provided in the table below.

Methods of the xhcidbg.XHCIDBG endpoint (kl.core.XHCIDBG interface)

Method	Method purpose and parameters	Potential danger of the method
`Suspend`	Purpose Suspends the XHCI DbC driver of the KasperskyOS kernel. Parameters • [out] `rc` – return code.	Configures the XHCI controller to not send diagnostic output (or debugging) through a USB port.
`Resume`	Purpose Starts a suspended XHCI DbC driver of the KasperskyOS kernel. Parameters • [out] `rc` – return code.	Allows the following: • Configure the XHCI controller so that diagnostic output (or debugging) is performed through the USB port. • Start a suspended XHCI DbC driver of the KasperskyOS kernel before restarting the XHCI controller, which will make it impossible to perform diagnostic output (or debugging) via the USB port after restarting the XHCI controller.

Audit endpoint

This endpoint is intended for reading the KasperskyOS kernel log containing security audit data.

Information about methods of the endpoint is provided in the table below.

Methods of the audit.Audit endpoint (kl.core.Audit interface)

Method	Method purpose and parameters	Potential danger of the method
`Open`	Purpose Opens the kernel log containing security audit data. Parameters • [in] `name` – name of the kernel log containing security audit data (`kss`). • [out] `handle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. This handle identifies the kernel log containing security audit data. • [out] `rc` – return code.	N/A
`Close`	Purpose Closes the kernel log containing security audit data. Parameters • [in] `handle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. This handle identifies the kernel log containing security audit data. • [out] `rc` – return code.	N/A
`Read`	Purpose Receives a message from the kernel log containing security audit data. Parameters • [in] `handle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. This handle identifies the kernel log containing security audit data. • [out] `msg` – sequence containing a message. • [out] `outDropMsgs` – number of messages that were lost due to overwrites in the kernel log containing security audit data. • [out] `rc` – return code.	Reads messages from the kernel log containing security audit data so that these messages are not received by another process.

Profiling endpoint

This endpoint is intended for profiling and collecting code coverage.

Information about methods of the endpoint is provided in the table below.

Methods of the profiler.Profiler endpoint (kl.core.Profiler interface)

Method	Method purpose and parameters	Potential danger of the method
`GetCoverageData`	Purpose Gets information about kernel code coverage. Parameters • [in] `index` – index for enumerating object files containing instrumental code for receiving coverage data. Enumeration starts with zero. • [out] `buf` – sequence containing information about the code coverage of an object file (in gcda format). • [out] `name` – name of the `*.gcda` file that was assigned during compilation. • [out] `rc` – return code.	N/A
`FlushGcov`	Purpose Output of data on kernel code coverage in gcda format via I/O interface (for example, COM, USB). Parameters • [out] `rc` – return code.	N/A
`FlushGcovFile`	Purpose Output of data on user code coverage in gcda format via I/O interface (for example, COM, USB). Parameters • [in] `name` – name of the `*.gcda` file that was assigned during compilation. • [in] `buf` – pointer to the buffer containing information about code coverage in gcda format. • [in] `size` – size of data containing code coverage information (in bytes). • [out] `rc` – return code.	N/A
`SamplingStart`	Purpose Starts sampling-based code profiling. Parameters • [in] `conf` – flags that define the profiling settings. • [in] `cpus` – CPU mask indicating the set of processors (processor cores) that execute the profiled code. • [in] `contSize` – size (in bytes) of the container used to store data containing the code execution statistics obtained from profiling. • [in] `interval` – fictitious parameter. • [out] `rc` – return code.	N/A

Method	Method purpose and parameters	Potential danger of the method
`SamplingStop`	Purpose Stops sampling-based code profiling. Parameters • [out] `rc` – return code.	N/A
`SamplingRead`	Purpose Gets data containing the code execution statistics received from sampling-based profiling. Parameters • [in] `unsafeBuffer` – pointer to the buffer used to save the data on the code execution statistics obtained from profiling. • [in] `size` – size of the buffer whose pointer is defined through the `unsafeBuffer` parameter (in bytes). • [out] `realSize` – size (in bytes) of the saved data. • [in] `timeout` – container filling timeout (in milliseconds). • [out] `rc` – return code.	Gets the addresses and names of functions of other processes.
`SamplingAddPidToList`	Purpose Adds a process to the set of processes in whose contexts the programs subject to sampling-based profiling are running. Parameters • [in] `pid` – process ID (PID). • [out] `rc` – return code.	N/A
`SamplingClearPidList`	Purpose Clears the set of processes in whose contexts the programs subject to sampling-based profiling are running. Parameters • [out] `rc` – return code.	N/A

Method	Method purpose and parameters	Potential danger of the method
`LoadSegInfo`	Purpose Saves information about the loaded segment of the ELF image in the kernel. Parameters • [in] `task` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the process. • [in] `addr` – segment address in process memory. • [in] `size` – segment size (in bytes). • [in] `offset` – offset of the segment in the ELF file (in bytes). • [in] `flags` – flags defining the access rights to the segment. • [in] `buildId` – build ID. • [out] `rc` – return code.	Allows the kernel memory to be used up by creating a multitude of objects within it.
`UnloadSegInfo`	Purpose Deletes information about the loaded ELF image segment that was saved in the kernel using the `LoadSegInfo` method. Parameters • [in] `task` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the process. • [in] `addr` – segment address in process memory. • [in] `size` – segment size (in bytes). • [out] `rc` – return code.	N/A
`KcovAlloc`	Purpose Allocates the resources required for collecting kernel code coverage data when handling system calls executed by the calling process. Parameters • [in] `numThreads` – maximum number of threads whose kernel code coverage data will be collected. • [in] `maxPoints` – maximum number of coverage points for one thread. • [out] `rc` – return code.	Exhausts RAM.
`KcovFree`	Purpose Frees the resources required for collecting kernel code coverage data when handling system calls executed by the calling process. Parameters • [out] `rc` – return code.	N/A

Method	Method purpose and parameters	Potential danger of the method
`KcovStart`	Purpose Starts the collection of kernel code coverage data when handling system calls executed by the calling thread. Parameters • [out] `rc` – return code.	N/A
`KcovStop`	Purpose Stops the collection of kernel code coverage data when handling system calls executed by the calling thread. Also gets information about kernel code coverage. Parameters • [in] `points` – pointer to the buffer used to store the IDs of coverage points. • [in] `maxPoints` – maximum number of coverage point IDs that can be stored in the buffer defined via the `points` parameter. • [out] `numPoints` – actual number of coverage point IDs stored in the buffer defined via the `points` parameter. • [out] `rc` – return code.	N/A

I/O memory isolation management endpoint

This endpoint is intended for managing the isolation of physical memory regions used by devices on a PCIe bus for DMA. (Isolation is provided by the IOMMU.)

Information about methods of the endpoint is provided in the table below.

Methods of the iommu.IOMMU endpoint (kl.core.IOMMU interface)

Method	Method purpose and parameters	Potential danger of the method
`Attach`	Purpose Attaches a device on a PCIe bus to the automatically created IOMMU domain associated with the calling process. Parameters • [in] `bdf` – address of the device on the PCIe bus in BDF format. • [out] `rc` – return code.	Attaches a device on a PCIe bus managed by another process to an IOMMU domain associated with the calling process, which leads to failure of the device. It is recommended to monitor the address of a device on a PCIe bus.
`Detach`	Purpose Detaches a device on a PCIe bus from the automatically created IOMMU domain associated with the calling process. Parameters • [in] `bdf` – address of the device on the PCIe bus in BDF format. • [out] `rc` – return code.	N/A

Method	Method purpose and parameters	Potential danger of the method
`CreateDomain`	Purpose Creates an IOMMU domain associated with the calling process. Parameters • [out] `domain` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the IOMMU domain. • [out] `rc` – return code.	Allows the kernel memory to be used up by creating a multitude of objects within it.
`AttachToDomain`	Purpose Attaches a device on a PCIe bus to the defined IOMMU domain associated with the calling process. Parameters • [in] `bdf` – address of the device on the PCIe bus in BDF format. • [in] `domain` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the IOMMU domain. • [out] `rc` – return code.	Attaches a device on a PCIe bus managed by another process to an IOMMU domain associated with the calling process, which leads to failure of the device. It is recommended to monitor the address of a device on a PCIe bus.
`DetachFromDomain`	Purpose Detaches a device on a PCIe bus from the defined IOMMU domain associated with the calling process. Parameters • [in] `bdf` – address of the device on the PCIe bus in BDF format. • [in] `domain` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the IOMMU domain. • [out] `rc` – return code.	N/A

Power management endpoint

This endpoint is intended for changing the power management mode of a hardware platform (for example, shutdown, restart, or hibernation), and for enabling and disabling processors (computing cores).

Information about methods of the endpoint is provided in the table below.

Methods of the pm.PM endpoint (kl.core.PM interface)

Method	Method purpose and parameters	Potential danger of the method
`Request`	Purpose Requests to change the power mode of a hardware platform. Parameters • [in] `request` – value defining the necessary power mode of the hardware platform. • [out] `rc` – return code.	Allows the hardware platform power mode to be changed.
`SetCpusOnline`	Purpose Requests to enable and/or disable processors. Parameters • [in] `request` – CPU mask indicating the required set of active processors. • [out] `rc` – return code.	Disables and enables processors.
`GetCpusOnline`	Purpose Gets information regarding which processors are in the active state. Parameters • [out] `online` – CPU mask indicating the actual set of active processors. • [out] `rc` – return code.	N/A

Notifications endpoint

This endpoint is intended for working with notifications about events associated with resources.

Information about methods of the endpoint is provided in the table below.

Methods of the notice.Notice endpoint (kl.core.Notice interface)

Method	Method purpose and parameters	Potential danger of the method
`Create`	Purpose Creates a notification receiver. Parameters • [out] `notify` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the notification receiver. • [out] `rc` – return code.	Allows the kernel memory to be used up by creating a multitude of objects within it.

Method	Method purpose and parameters	Potential danger of the method
`SubscribeToObject`	Purpose Adds a "resource–event mask" entry to the notification receiver so that it can receive notifications about events associated with the defined resource and match the defined event mask. Parameters • [in] `notify` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the notification receiver. • [in] `object` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the resource. • [in] `evMask` – event mask. • [in] `evId` – ID of the "resource–event mask" entry. • [in] `flags` – flags defining the parameters for the "resource–event mask" entry. • [out] `rc` – return code.	Allows the kernel memory to be used up by creating a multitude of objects within it.
`UnsubscribeFromEvent`	Purpose Removes from the notification receiver "resource—event mask" entries with the specified identifier to prevent the receiver from getting notifications about events that match these entries. Parameters • [in] `notify` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the notification receiver. • [in] `evId` – ID of the "resource–event mask" entries. • [in] `object` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the resource. If the resource is specified, the method deletes the "resource–event mask" entries that have the ID specified in the `evId` parameter and correspond to this resource. • [out] `rc` – return code.	N/A
`UnsubscribeFromObject`	Purpose Removes from the notification receiver "resource—event mask" entries that match the specified resource to prevent the receiver from getting notifications about events that match these entries. Parameters • [in] `notify` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the notification receiver. • [in] `object` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the resource. • [in] `flags` – flags defining which "resource–event mask" entries should be deleted. • [out] `rc` – return code.	N/A

Method	Method purpose and parameters	Potential danger of the method
`GetEvent`	Purpose Extracts notifications from the receiver. Parameters • [in] `notify` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the notification receiver. • [in] `mdelay` – timeout (in milliseconds) for notifications to appear in the receiver. • [out] `events` – sequence of notifications comprised of structures containing a "resource–event mask" entry ID, mask and counter of events associated with the resource. • [out] `rc` – return code.	N/A
`DropAndWake`	Purpose Resumes execution of all threads that are waiting for notifications to appear in the receiver and performs additional actions depending on the `flags` parameter: • Denies extraction of notifications from the receiver. • Defines the method used to reallow extraction of notifications from the receiver. • Deletes all "resource–event mask" entries and already received notifications from the receiver. • Disables the notification receiver. Parameters • [in] `notify` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the notification receiver. • [in] `flags` – flags defining additional actions. • [out] `rc` – return code.	N/A
`SetObjectEvent`	Purpose Signals the occurrence of events that are related to the defined user resource and match the defined event mask. Parameters • [in] `object` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the user resource. • [in] `evMask` – mask of events to be signaled. • [out] `rc` – return code.	N/A

Method	Method purpose and parameters	Potential danger of the method
`EditSubscriptionForEvent`	Purpose Modifies "resource–event mask" entries with the specified ID in the notification receiver. Parameters • [in] `notify` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the notification receiver. • [in] `evId` – ID of the "resource–event mask" entries that need to be modified. • [in] `object` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the resource. If the resource is specified, the method modifies the "resource–event mask" entries that have the ID specified in the `evId` parameter and correspond to this resource. • [in] `newId` – new ID of the "resource–event mask" entries. • [in] `newMask` – new event mask. • [out] `rc` – return code.	N/A
`EditSubscriptionForObject`	Purpose Modifies the "resource–event mask" entries in the notification receiver that correspond to the specified resource. Parameters • [in] `notify` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the notification receiver. • [in] `object` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the resource. • [in] `flags` – flags defining which "resource–event mask" entries should be modified. • [in] `newId` – new ID of the "resource–event mask" entries. • [in] `newMask` – new event mask. • [out] `rc` – return code.	N/A
`Resume`	Purpose Reallows extraction of notifications from the receiver denied by the `DropAndWake` method. Parameters • [in] `notify` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the notification receiver. • [out] `rc` – return code.	N/A

Trusted Execution Environment endpoint

This endpoint is intended for transferring data between a *Trusted Execution Environment* (TEE) and a *Rich Execution Environment* (REE).

Information about methods of the endpoint is provided in the table below.

Methods of the tee.TEE endpoint (kl.core.TEE interface)

Method	Method purpose and parameters	Potential danger of the method
`Dispatch`	Purpose Sends and receives messages transferred between a TEE and a REE. This method is used in the TEE and in the REE. Parameters • [in] `msgIn` – structure containing a request for the TEE (when the method is called in the REE) or a response for the REE (when the method is called in the TEE). • [out] `msgOut` – structure containing a response from the TEE (when the method is called in the REE) or a request from the REE (when the method is called in the TEE). • [out] `rc` – return code.	Allows a process in a REE to receive a response from a TEE regarding a request from another process in the REE.
`FreeToken`	Purpose Frees the values of unique IDs of messages transferred between a TEE and a REE. (These values must be freed so that they can become available for re-use.) This method is used in REE. Parameters • [in] `token` – value of the unique ID of a message. • [out] `rc` – return code.	Frees the values used by other processes in a REE as unique IDs of messages transferred between a TEE and a REE.

IPC management endpoint

This endpoint is intended for interrupting the `Call()` and `Recv()` locking system calls. (For example, this may be required to correctly terminate a process.)

Information about methods of the endpoint is provided in the table below.

Methods of the ipc.IPC endpoint (kl.core.IPC interface)

Method	Method purpose and parameters	Potential danger of the method
`CreateSyncObject`	Purpose Creates an IPC synchronization object. An IPC synchronization object is used to interrupt `Call()` and `Recv()` locking system calls in threads of the calling process. Parameters • [in] `autoclear` – value that indicates whether or not you need to create an IPC synchronization object that automatically switches to a state in which the `Call()` and `Recv()` system calls are not interrupted after the first interruption of these system calls (a non-zero value means yes and `0` means no). • [out] `syncHandle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the IPC synchronization object. • [out] `rc` – return code.	Allows the kernel memory to be used up by creating a multitude of objects within it.
`SetInterrupt`	Purpose Switches the defined IPC synchronization object to a state in which the `Call()` and `Recv()` system calls are interrupted. Parameters • [in] `syncHandle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the IPC synchronization object. • [out] `rc` – return code.	N/A
`ClearInterrupt`	Purpose Switches the defined IPC synchronization object to a state in which the `Call()` and `Recv()` system calls are not interrupted. Parameters • [in] `syncHandle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the IPC synchronization object. • [out] `rc` – return code.	N/A

Hardware platform firmware communication endpoint

The endpoint is intended for interaction with firmware of the hardware platform via the UEFI interface.

Information about methods of the endpoint is provided in the table below.

Methods of the efi.Efi endpoint (kl.core.Efi interface)

Method	Method purpose and parameters	Potential danger of the method
`GetTime`	Purpose Lets you get the current time from the time source on the hardware platform and the characteristics of this source. Parameters • [out] `time` – structure containing time information. • [out] `capabilities` – structure containing the characteristics of the time source. • [out] `rc` – return code.	N/A
`SetTime`	Purpose Sets the current time in the time source on the hardware platform. Parameters • [in] `time` – structure containing time information. • [out] `rc` – return code.	Lets you set the current time in the time source on the hardware platform.
`GetVariable`	Purpose Gets the value of the UEFI variable. Parameters • [in] `variableName` – sequence containing the name of a variable with a terminating zero. • [in] `vendorGuid` – structure containing the unique ID of the vendor related to the variable. • [out] `attributes` – value whose bits are interpreted as attributes of a variable. • [out] `dataSizeReal` – size (bytes) of data received via the `data` parameter. • [out] `data` – sequence containing the value of the variable. • [out] `rc` – return code.	Gets values of the UEFI variables that could contain critical system information.

Method	Method purpose and parameters	Potential danger of the method
`GetNextVariableName`	Purpose Enumerates the names of the UEFI variables. Parameters • [in] `variableNamePrev` – sequence containing the name of the variable with a terminating zero obtained during the previous call of the `GetNextVariableName` method, or null if the `GetNextVariableName` method is being called for the first time. • [in] `vendorGuidPrev` – structure containing the unique ID of the vendor related to the variable whose name was received during the previous call of the `GetNextVariableName` method, or arbitrary values if a sequence of zeros is passed in the `variableNamePrev` parameter. • [out] `variableNameSizeReal` – size (bytes) of data received via the `variableNameNext` parameter. • [out] `variableNameNext` – sequence containing the name of a variable with a terminating zero. • [out] `vendorGuidNext` – structure containing the unique ID of the vendor related to the variable. • [out] `rc` – return code.	Gets the names of the UEFI variables that could contain critical system information.
`SetVariable`	Purpose Specifies the value of the UEFI variable. Parameters • [in] `variableName` – sequence containing the name of a variable with a terminating zero. • [in] `vendorGuid` – structure containing the unique ID of the vendor related to the variable. • [in] `attributes` – value whose bits are interpreted as attributes of the variable. • [in] `data` – sequence containing the value of the variable. • [out] `rc` – return code.	Lets you set the value of the UEFI variable.
`QueryVariableInfo`	Purpose Gets information about UEFI variables. Parameters • [in] `attributes` – value whose bits are interpreted as attributes of the variables whose information you need to get. Each corresponding variable must have the entire set of defined attributes (excluding those that are ignored by the UEFI specification). • [out] `maximumVariableStorageSize` – size (in bytes) of memory that is intended for storing variables with specific attributes. • [out] `remainingVariableStorageSize` – size (in bytes) of the unused portion of memory that is intended for storing variables with specific attributes. • [out] `maximumVariableSize` – maximum size (in bytes) of variables with specific attributes. • [out] `rc` – return code.	N/A

Method	Method purpose and parameters	Potential danger of the method
`ResetSystem`	Purpose Restarts the hardware platform. Parameters • [in] `resetType` – value that defines the restart type, such as cold or hot restart. • [in] `resetStatus` – code that indicates the restart status, such as normal restart or restart due to malfunction. • [in] `resetData` – sequence containing information about the causes of restarts. • [out] `rc` – return code.	Lets you restart the hardware platform.

Statistics endpoint

This endpoint is intended for receiving values of performance counters.

Information about methods of the endpoint is provided in the table below.

Methods of the stat.Stat endpoint (kl.core.Stat interface)

Method	Method purpose and parameters	Potential danger of the method
`GetStatValues`	Purpose Gets the values of performance counters. Parameters • [in] `prefix` – prefix for names of performance counters. • [in] `names` – sequence containing the names of performance counters. • [out] `values` – sequence containing the values of performance counters. • [out] `rc` – return code.	N/A
`GetObjectStatValues`	Purpose Gets the values of performance counters associated with a system resource (process or thread). Parameters • [in] `handle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the system resource. • [in] `names` – sequence containing the names of performance counters. • [out] `values` – sequence containing the values of performance counters. • [out] `rc` – return code.	N/A

BLOB endpoint

The endpoint is designed to gain access to named blobs (for example, access to the ROMFS image that is part of the KasperskyOS kernel image).

Information about methods of the endpoint is provided in the table below.

Methods of the blob.Blob endpoint (kl.core.Blob interface)

Method	Method purpose and parameters	Potential danger of the method
`Get`	Purpose Gets the handle of a blob based on the name of this blob. Parameters • [in] `name` – BLOB name. • [out] `handle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the blob. • [out] `size` – size of the BLOB in bytes. • [out] `rc` – return code.	Allows the kernel memory to be used up by creating a multitude of objects within it.
`Map`	Purpose Reserves a virtual memory region and maps the BLOB to it. Parameters • [in] `resource` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the blob. • [in] `prot` – flags defining the access rights to the BLOB. • [in] `attr` – flags defining the parameters of the virtual memory region. • [out] `address` – base address of the virtual memory region. • [out] `mapping` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the virtual memory region. • [out] `rc` – return code.	Allows the kernel memory to be used up by creating a multitude of objects within it.
`GetSize`	Purpose Gets the size of the BLOB. Parameters • [in] `handle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the blob. • [out] `size` – size of the BLOB in bytes. • [out] `rc` – return code.	N/A

Method	Method purpose and parameters	Potential danger of the method
`Remove`	Purpose Initiates deletion of a blob. Parameters • [in] `name` – BLOB name. • [out] `rc` – return code.	Deletes blobs.

Endpoint for adapting the scheduler to a hardware platform that uses processors of varying performance

The endpoint is designed to set the performance levels of processors (computational cores) in the scheduler.

Information about methods of the endpoint is provided in the table below.

Methods of the sched.Sched endpoint (kl.core.Sched interface)

Method	Method purpose and parameters	Potential danger of the method
`SetCpuCapacity`	Purpose Defines the processor performance level in the scheduler. Parameters • [in] `cpu` – processor index. • [in] `capacity` – processor performance level. • [out] `rc` – return code.	Defines the processor performance levels in the scheduler so that high-performance processors are used as low-performance processors and/or low-performance processors are used as high-performance processors.

Process diagnostics endpoint

The endpoint is designed to receive build IDs for ELF images (executable file and dynamic link libraries) loaded into process memory.

Information about methods of the endpoint is provided in the table below.

Methods of the taskDebug.TaskDebug endpoint (kl.core.TaskDebug interface)

Method	Method purpose and parameters	Potential danger of the method
`GetBuildId`	Purpose Gets the build ID for the ELF image whose segment occupies a region in the memory of the calling process that includes the specified address. Parameters • [in] `addr` – address within the memory region of the process occupied by the ELF image segment. • [out] `buildId` – sequence containing the build ID. • [out] `rc` – return code.	N/A

Security patterns for development under KasperskyOS

Each KasperskyOS-based solution has specific usage scenarios and is designed to counteract specific security threats. Nonetheless, there are some typical scenarios and threats encountered in many different solutions. This section describes the typical risks and threats, and contains a description of architectural patterns that can be employed to increase the security of a solution.

A *security pattern (or template)* describes a specific recurring security issue that arises in certain known contexts, and provides a well-proven, general scheme for resolving this kind of security issue. A pattern is not a finished project that can be converted directly into code. Instead, it is a solution to a general problem encountered in various projects.

A *security pattern system* is a set of security patterns together with instructions on their implementation, combination, and practical use when designing secure software systems.

Security patterns resolve security issues at different levels, beginning with patterns at the architectural level, including high-level design of the system, and ending with implementation-level patterns that contain recommendations on how to implement functions or methods.

This section describes the set of security patterns whose implementation examples are provided in KasperskyOS Community Edition SDK.

> Security patterns are described in a multitude of information security resources. Each pattern is accompanied by a list of the resources that were used to prepare its description.

Distrustful Decomposition pattern

Description

When using a monolithic application, a single process must be granted all the privileges necessary for the application to operate. This issue is resolved by the `Distrustful Decomposition` pattern.

The purpose of the `Distrustful Decomposition` pattern is to divide application functionality among individual processes that require different levels of privileges, and to control the interaction between these processes instead of creating a monolithic application.

Using the `Distrustful Decomposition` pattern reduces the following:

- Attack surface for each process.
- Functionality and data that a hacker will be able to access if one of the processes is compromised.

Alternate names

`Privilege Reduction`.

Context

Different functions of an application require different levels of privileges.

Problem

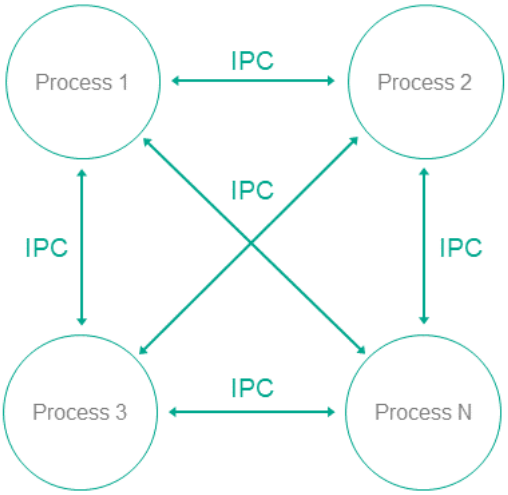
An unsophisticated implementation of an application combines many functions requiring different privileges into one component. This component would need to be run with the maximum level of privileges required for any one of these many functions.

Solution

The `Distrustful Decomposition` pattern divides functionality among individual processes and isolates potential vulnerabilities within a small subset of the system. A cybercriminal who conducts a successful attack will be able to use only the functionality and data of a single compromised component instead of the entire application.

Structure

This pattern divides one monolithic application into multiple applications that are run as individual processes that could potentially have different privileges. Each process implements a small, clearly defined set of functions of the application. Processes use interprocess communication mechanism to exchange data.



Operation

- In KasperskyOS, an application is divided into processes.
- Processes can exchange messages via IPC.
- A user or remote system connects to the process that provides the necessary functionality with the level of privileges sufficient to perform the requested functions.

Implementation recommendations

Interaction between processes can be unidirectional or bidirectional. It is recommended to always use unidirectional interaction whenever possible. Otherwise, the potential attack surface of individual components increases, which reduces the overall security of the entire system. If bidirectional IPC is used, processes should not trust bidirectional data exchange. For example, if a file system is used for IPC, file contents cannot be trusted.

Specialized implementation in KasperskyOS

In universal operating systems such as Linux or Windows, this pattern does not use anything except the standard process/privileges model that already exists in these operating systems. Each program is run in its own process space with potentially different privileges of the specific user in each process. However, an attack on the OS kernel would reduce the effectiveness of this pattern.

Use of this pattern when developing for KasperskyOS means that control over processes and IPC is entrusted to the microkernel, which is difficult to successfully attack. The Kaspersky Security Module is used for IPC control.

Use of KasperskyOS mechanisms ensures a high level of reliability of the software system with the same or less effort required from the developer when compared to the use of this pattern in programs running under universal operating systems.

In addition, KasperskyOS provides the capability for flexible configuration of security policies. Moreover, the process of defining and editing security policies is potentially independent of the process of developing the applications.

Linked patterns

Use of the `Distrustful Decomposition` pattern involves use of the Defer to Kernel and Policy Decision Point patterns.

Implementation examples

Examples of an implementation of the `Distrustful Decomposition` pattern:

- Secure Logger
- Separate Storage

Sources of information

The `Distrustful Decomposition` pattern is described in detail in the following resources:

- Chad Dougherty, Kirk Sayre, Robert C. Seacord, David Svoboda, Kazuya Togashi (JPCERT/CC), "Secure Design Patterns" (March-October 2009). Software Engineering Institute. https://resources.sei.cmu.edu/asset_files/TechnicalReport/2009_005_001_15110.pdf
- Dangler, Jeremiah Y., "Categorization of Security Design Patterns" (2013). Electronic Theses and Dissertations. Paper 1119. https://dc.etsu.edu/etd/1119

Secure Logger example

The `Secure Logger` example demonstrates use of the Distrustful Decomposition pattern for separating event log read/write functionality.

Example architecture

The security objective of the `Secure Logger` example is to prevent any possibility of distortion or deletion of information from the event log. This example utilizes the capabilities provided by KasperskyOS to achieve this security objective.

A logging system can be examined by distinguishing the following functional steps:

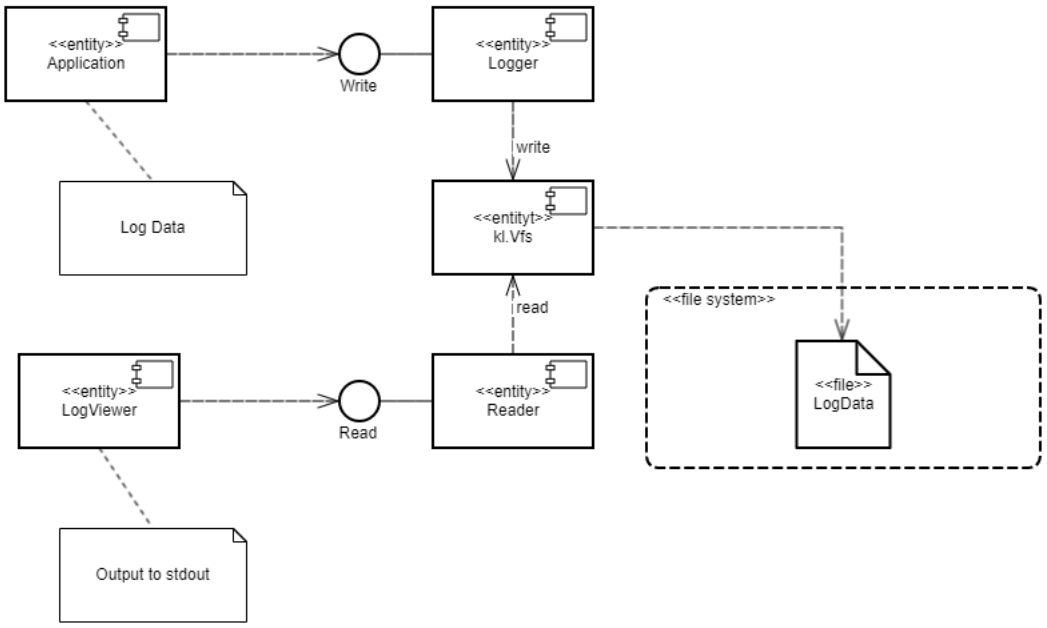
- Generate information to be written to the log.
- Save information to the log.
- Read entries from the log.
- Provide entries in a convenient format for the consumer.

Accordingly, the logging subsystem can be divided into four processes depending on the required functional capabilities of each process.

For this purpose, the `Secure Logger` example contains the following four programs: `Application`, `Logger`, `Reader` and `LogViewer`.

- The `Application` program initiates the creation of entries in the event log maintained by the `Logger` program.
- The `Logger` program creates entries in the log and writes them to the disk.
- The `Reader` program reads entries from the disk to send them to the `LogViewer` program.
- The `LogViewer` program sends entries to the user.

The IPC interface provided by the `Logger` program is intended *only* for writing to storage. The IPC interface of the `Reader` program is intended only for reading from storage. The example architecture looks as follows:



- The `Application` program uses the interface of the `Logger` program to save log entries.
- The `LogViewer` program uses the interface of the `Reader` program to read the log entries and present them to a user.

The `LogViewer` program normally has external channels for interacting with a user (for example, to receive data write commands and to provide data to a user). Naturally, this program is an untrusted component of the system, and therefore could potentially be used to conduct an attack. However, even if a successful attack results in the infiltration of unauthorized executable code into the `LogViewer` program, information in the log cannot be distorted through this program. This is because the program can only utilize the data read interface, which cannot actually be used to distort or delete data. Moreover, the `LogViewer` program does not have the capability to gain access to other interfaces because this access is controlled by the security module.

A security policy in the `Secure Logger` example has the following characteristics:

- The `Application` program has the capability to query the `Logger` program to create a new entry in the event log.
- The `LogViewer` program has the capability to query the `Reader` program to read entries from the event log.

- The `Application` program *does not* have the capability to query the `Reader` program to read entries from the event log.
- The `LogViewer` program *does not* have the capability to query the `Logger` program to create a new entry in the event log.

Dynamically created IPC channels

IPC connections between programs are established dynamically using the DCM system program, in which servers publish their interfaces and clients can connect to previously published interfaces. Please note that the same program can act as a client and server at the same time and interact with DCM as both a client and a server.

Example files

The code of the example and build scripts are available at the following path:

```
/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples/secure_logger
```

Building and running the example

See Building and running examples section.

Separate Storage example

The `Separate Storage` example demonstrates use of the Distrustful Decomposition pattern to separate data storage for trusted and untrusted applications.

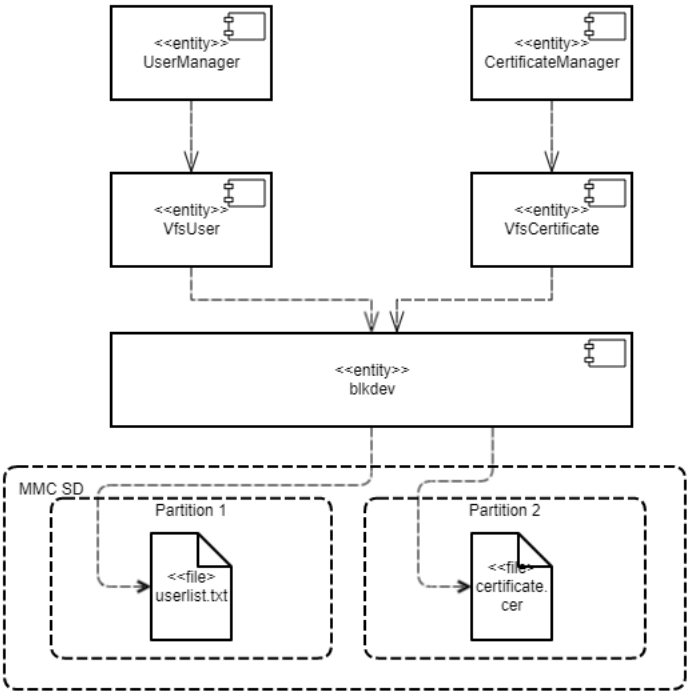
Example architecture

The `Separate Storage` example contains two user programs: `UserManager` and `CertificateManager`.

These programs work with data located in the corresponding files:

- The `UserManager` program works with data from the `userlist.txt` file.
- The `CertificateManager` program works with data from the `certificate.cer` file.

Each of these programs uses its own instance of the VFS program to access a separate file system. Each VFS program includes a block device driver linked to an individual logical drive partition. The `UserManager` program does not have access to the file system of the `CertificateManager` program, and vice versa.



This architecture guarantees that if there is an attack or error in any of the `UserManager` or `CertificateManager` programs, this program will not be able to access any file that was not intended for the specific program's operations.

A security policy in the `Separate Storage` example has the following characteristics:

- The `UserManager` program has access to the file system *only* through the `VfsUser` program.
- The `CertificateManager` program has access to the file system *only* through the `VfsCertificate` program.

Example files

The code of the example and build scripts are available at the following path:

```
/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples/separate_storage
```

Building and running the example

The `CMake` system, which is included with KasperskyOS Community Edition SDK, is used to build and run the example.

See Building and running examples section.

Defer to Kernel pattern

Description

The `Defer to Kernel` pattern takes advantage of permission control at the OS kernel level.

The purpose of this pattern is to utilize mechanisms available at the OS kernel level to clearly separate the functionality requiring elevated privileges from the functionality that does not require elevated privileges. By using kernel mechanisms, we do not have to implement new tools for arbitrating security decisions at the user level.

Alternate names

`Policy Enforcement Point (PEP)`, `Protected System`, `Enclave`.

Context

The `Defer to Kernel` pattern is applicable if the system has the following characteristics:

- The system has processes that run without elevated privileges, including user processes.
- Some system functions require elevated privileges that must be verified before processes are granted access to data.
- You need to verify not only the privileges of the requesting process, but also the overall permissibility of the requested operation within the operational context of the entire system and its overall security.

Problem

When functionality is divided among various processes with different levels of privileges, these privileges must be verified when a request is made from one process to another. These verifications must be carried out and their resulting permissions must be granted by trusted code that has a minimal risk of being compromised. The trustworthiness of application code is almost always questionable due to its sheer volume and due to its primary orientation toward implementation of functional requirements.

Solution

Clearly separate privileged functionality and data from non-privileged functionality and data at the process level, and give the OS kernel control of interprocess communication (IPC), including verification of access rights when there is a request for functionality or data requiring elevated privileges, and verification of the overall state of the system and the states of individual processes at the time of the request.

Structure



Operation

- Functionality and management of data with various privileges are compartmentalized among processes.
- The OS kernel ensures isolation of processes.
- `Process-1` wants to request privileged functionality or data from `Process-2` using IPC.
- The kernel controls IPC and allows or denies communication based on security policies and based on the available information regarding the operational context and state of `Process-1`.

Implementation recommendations

To ensure that a specific implementation of a pattern operates securely and reliably, the following is required:

- **Isolation**

  Complete and guaranteed isolation of processes must be ensured.

- **Inability to bypass the kernel**

  Absolutely all IPC interactions must be controlled by the kernel.

- **Kernel self-defense**

  The trustworthiness of the kernel must be ensured through its own means of protection against compromise.

- **Provability**

  The kernel requires a certain level of guaranteed security and reliability.

- **Capability for external computation of access permissions**

  Access permissions must be computed at the OS level, and must not be implemented in application code.

  For this purpose, tools must be provided for describing access policies so that security policies are detached from the business logic.

Specialized implementation in KasperskyOS

The KasperskyOS kernel guarantees isolation of processes and serves as a Policy Enforcement Point (PEP).

Linked patterns

The `Defer to Kernel` pattern is a special case of the Distrustful Decomposition and Policy Decision Point patterns. The `Policy Decision Point` pattern defines the abstraction process that intercepts all requests to resources and verifies that they comply with the defined security policy. The distinctive feature of the `Defer to Kernel` pattern is that the verification process is performed by the OS kernel, which is a more reliable and portable solution that reduces the time spent on development and testing.

Impacts

By making the OS kernel responsible for applying the access policy, you separate the security policy from the business logic (which may be very complicated) and thereby simplify development and improve portability through the use of OS kernel functions.

This also makes it possible to prove the overall security of a solution by simply demonstrating that the kernel is operating correctly. The difficulty in proving correct execution of code grows nonlinearly as the size of the code increases. The `Defer to Kernel` pattern minimizes the amount of trusted code, provided that the OS kernel itself is not too large.

Implementation examples

Example of a `Defer to Kernel` pattern implementation: Defer to Kernel example.

Sources of information

The `Defer to Kernel` pattern is described in detail in the following resources:

- Chad Dougherty, Kirk Sayre, Robert C. Seacord, David Svoboda, Kazuya Togashi (JPCERT/CC), "Secure Design Patterns" (March-October 2009). Software Engineering Institute. https://resources.sei.cmu.edu/asset_files/TechnicalReport/2009_005_001_15110.pdf
- Dangler, Jeremiah Y., "Categorization of Security Design Patterns" (2013). Electronic Theses and Dissertations. Paper 1119. https://dc.etsu.edu/etd/1119
- Schumacher, Markus, Fernandez-Buglioni, Eduardo, Hybertson, Duane, Buschmann, Frank, and Sommerlad, Peter. "Security Patterns: Integrating Security and Systems Engineering" (2006)

Defer to Kernel example

The `Defer to Kernel` example demonstrates the use of Defer to Kernel and Policy Decision Point patterns.

The `Defer to Kernel` example contains three user programs: `PictureManager`, `ValidPictureClient` and `NonValidPictureClient`.

In this example, the `ValidPictureClient` and `NonValidPictureClient` programs query the `PictureManager` program to receive information.

Only the `ValidPictureClient` program is allowed to interact with the `PictureManager` program.

The KasperskyOS kernel guarantees isolation of running programs (processes).

Control of interaction between programs in KasperskyOS is delegated to the Kaspersky Security Module. The subsystem analyzes each sent request and response and decides whether to allow or deny delivery based on the defined security policy.

A security policy in the `Defer to Kernel` example has the following characteristics:

- The `ValidPictureClient` program is explicitly allowed to interact with the `PictureManager` program.
- The `NonValidPictureClient` program is explicitly *not* allowed to interact with the `PictureManager` program. This means that this interaction is denied (based on the *Default Deny principle*).

Dynamically created IPC channels

IPC connections between programs are established dynamically using the DCM system program, in which servers publish their interfaces and clients can connect to previously published interfaces. Please note that the same program can act as a client and server at the same time and interact with DCM as both a client and a server.

Example files

The code of the example and build scripts are available at the following path:

```
/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples/defer_to_kernel
```

Building and running the example

See Building and running examples section.

Policy Decision Point pattern

Description

The `Policy Decision Point` pattern encapsulates the computation of decisions based on security model methods into a separate system component that ensures that these security methods are performed in their full scope and correct sequence.

Alternate names

`Check Point`, `Access Decision Function`.

Context

The system has functions with different levels of privileges, and the security policy is complex (contains many security model methods bound to security events).

Problem

If security policy checks are divided among different system components, the following issues arise:

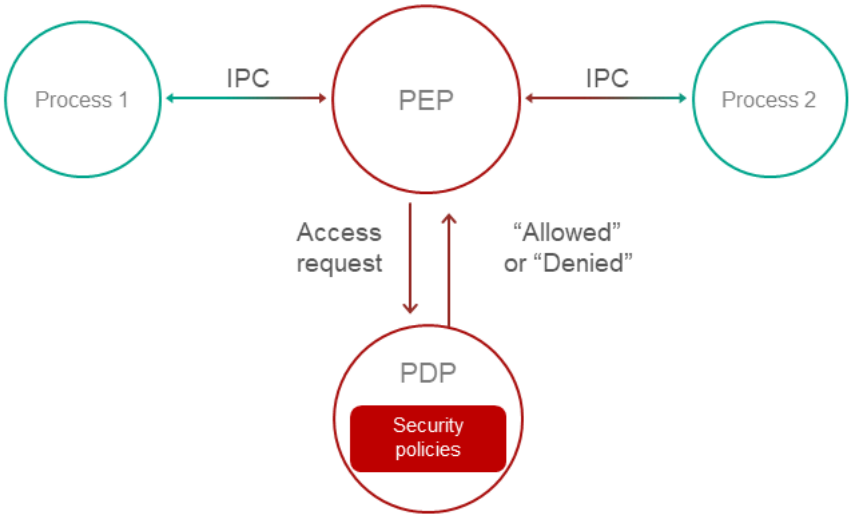
- You have to carefully make sure that all necessary checks are performed in all required cases.
- It is difficult to ensure that all checks are performed in the correct order.

- It is difficult to prove that the verification system is operating correctly, has no conflicts, and its integrity has not been compromised.
- The security policy is linked to the business logic. This means that any modification of the security policy requires changes to the business logic, which complicates support and increases the likelihood of errors.

Solution

All verifications of security policy compliance are conducted in a separate component called a Policy Decision Point (PDP). This component is responsible for ensuring that verifications are conducted in their correct sequence and scope. Policy checks are separated from the code that implements the business logic.

Structure



Operation

- A Policy Enforcement Point (PEP) receives a request to access functionality or data.

  For example, the PEP may be the OS kernel. For more details, refer to Defer to Kernel pattern.
- The PEP gathers the request attributes required for making decisions on access control.
- The PEP requests an access control decision from the Policy Decision Point (PDP).
- The PDP computes a decision on whether to grant access based on the security policy and based on the information received in the request from the PEP.
- The PEP denies or allows interaction based on the decision of the PDP.

Implementation recommendations

Implementations must take into account the problem of "Verification time vs. Usage time". For example, if a security policy depends on the quickly changing status of a specific system object, a computed decision loses its relevance as quickly as the status changes. In a system that utilizes the **Policy Decision Point** pattern, you must take care to minimize the time interval between the access decision and the time when the request based on this decision is fulfilled.

Specialized implementation in KasperskyOS

The KasperskyOS kernel guarantees isolation of processes and serves as a Policy Enforcement Point (PEP).

Control of interaction between processes in KasperskyOS is delegated to the Kaspersky Security Module. This module analyzes each sent request and response and decides whether to allow or deny delivery based on the defined security policy. Therefore, the Kaspersky Security Module performs the role of the Policy Decision Point (PDP).

Impacts

This pattern configures a security policy without making any modifications to the code that implements the business logic, and delegates system support involving information security.

Linked patterns

Use of the `Policy Decision Point` pattern involves use of the Distrustful Decomposition and Defer to Kernel patterns.

Implementation examples

Example of a `Policy Decision Point` pattern implementation: Defer to Kernel example.

Sources of information

The `Policy Decision Point` pattern is described in detail in the following resources:

- Chad Dougherty, Kirk Sayre, Robert C. Seacord, David Svoboda, Kazuya Togashi (JPCERT/CC), "Secure Design Patterns" (March-October 2009). Software Engineering Institute. https://resources.sei.cmu.edu/asset_files/TechnicalReport/2009_005_001_15110.pdf
- Dangler, Jeremiah Y., "Categorization of Security Design Patterns" (2013). Electronic Theses and Dissertations. Paper 1119. https://dc.etsu.edu/etd/1119
- Schumacher, Markus, Fernandez-Buglioni, Eduardo, Hybertson, Duane, Buschmann, Frank, and Sommerlad, Peter. "Security Patterns: Integrating Security and Systems Engineering" (2006)
- Bob Blakley, Craig Heath, and members of The Open Group Security Forum. "Security Design Patterns" (April 2004). The Open Group. https://pubs.opengroup.org/onlinepubs/9299969899/toc.pdf

Privilege Separation pattern

Description

The `Privilege Separation` pattern involves the use of non-privileged isolated system modules for interaction with clients (other modules or users) that do not have any privileges. The purpose of the `Privilege Separation` pattern is to reduce the amount of code that is executed with special privileges without impacting or restricting application functionality.

The `Privilege Separation` pattern is a special case of the Distrustful Decomposition pattern.

Example

An unauthenticated user connects to a system that has functions requiring elevated privileges.

Context

The system has components with a large attack surface due to their high number of connections with unsafe sources and/or a complicated, potentially error-prone implementation.

Problem

When a client with unknown privileges interacts with a privileged component of the system, there are risks that the data and functionality accessible to that component could be compromised.

Solution

Interactions with unsafe clients must be conducted only through specially allocated components that have no privileges. The `Privilege Separation` pattern does not modify system functionality. Instead, it merely separates functionality into components with different privileges.

Operation

Pattern operations can be divided into two phases:

- **Pre-Authentication.** The client is not yet authenticated. It sends a request to a privileged primary process. The primary process creates a child process with no privileges (and no access to the file system). This child process performs client authentication.
- **Post-Authentication.** The client is authenticated and authorized. The privileged primary process creates a new child process that has privileges corresponding to the permissions of the client. This process is responsible for all subsequent interaction with the client.

Recommendations on implementation in KasperskyOS

At the **Pre-Authentication** phase, the primary process can save the state of each non-privileged process in the form of a finite-state machine and change the state of the finite-state machine during authentication.

Requests from child processes to the primary process are performed using standard IPC mechanisms. However, interaction control is conducted using the Kaspersky Security Module.

Impacts

If attackers gain control of a non-privileged process, they will not gain access to any privileged functions or data. If attackers gain control of an authorized process, they will obtain only the privileges of this process.

In addition, code that is organized in this manner is easier to check and test. You just have to pay special attention to the functionality that operates with elevated privileges.

Implementation examples

Example of a `Privilege Separation` pattern implementation: Device Access example.

Sources of information

The `Privilege Separation` pattern is described in detail in the following resources:

- Chad Dougherty, Kirk Sayre, Robert C. Seacord, David Svoboda, Kazuya Togashi (JPCERT/CC), "Secure Design Patterns" (March-October 2009). Software Engineering Institute. https://resources.sei.cmu.edu/asset_files/TechnicalReport/2009_005_001_15110.pdf
- Dangler, Jeremiah Y., "Categorization of Security Design Patterns" (2013). Electronic Theses and Dissertations. Paper 1119. https://dc.etsu.edu/etd/1119

Device Access example

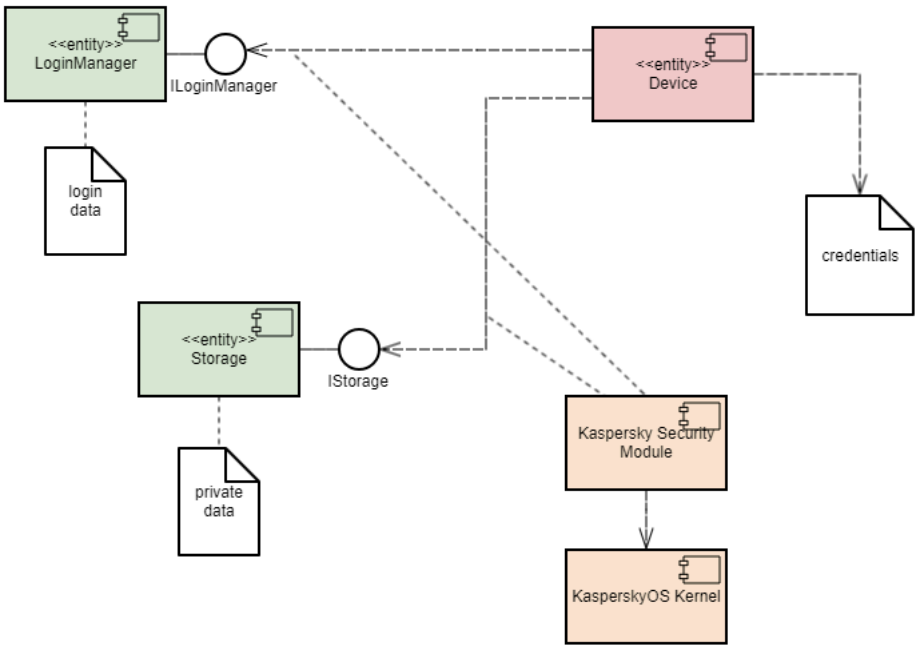
The `Device Access` example demonstrates use of the Privilege Separation pattern.

Example architecture

The example contains the following three programs: `Device`, `LoginManager` and `Storage`.

In this example, the `Device` program queries the `Storage` program to receive information and queries the `LoginManager` program for authorization.

The `Device` program obtains access to the `Storage` program after successful authorization.



This example demonstrates the capability to separate the authorization logic and the data access logic into independent components. This separation guarantees that data access can be opened only after successful authorization. The security module monitors whether authorization was successfully completed. This architecture also enables independent development and testing of the authorization logic and the data access provision logic.

A security policy in the `Device Access` example has the following characteristics:

- The `Device` program has the capability to query the `LoginManager` program for authorization.
- Calls of the `GetInfo()` method of the `Storage` program are managed by methods of the Flow security model:
  - The finite-state machine described in the `session` object configuration has two states: `unauthenticated` and `authenticated`.
  - The initial state is `unauthenticated`.
  - Only transitions from `unauthenticated` to `authenticated` and vice versa are allowed.
  - The `session` object is created when the `Device` program is started.
  - When the `Device` program successfully calls the `Login()` method of the `LoginManager` program, the state of the `session` object changes to `authenticated`.
  - When the `Device` program successfully calls the `Logout()` method of the `LoginManager` program, the state of the `session` object changes to `unauthenticated`.
  - When the `Device` program calls the `GetInfo()` method of the `Storage` program, the current state of the `session` object is verified. The call is allowed only if the current state of the object is `authenticated`.

Dynamically created IPC channels

IPC connections between programs are established dynamically using the DCM system program, in which servers publish their interfaces and clients can connect to previously published interfaces. Please note that the same program can act as a client and server at the same time and interact with DCM as both a client and a server.

Example files

The code of the example and build scripts are available at the following path:

```
/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples/device_access
```

Building and running the example

See Building and running examples section.

Information Obscurity pattern

Description

The purpose of the `Information Obscurity` pattern is to encrypt confidential data in otherwise unsafe environments and thereby protect against data theft.

Context

This pattern should be used when data is frequently transferred between parts of a system and/or between the system and other (external) systems.

Problem

Confidential data may be transmitted through an untrusted environment within one system (through untrusted components) or between different systems (through untrusted networks). If this environment is compromised, confidential data could be intercepted by a cybercriminal.

Solution

Data must be separated based on its specific level of confidentiality so that you can determine which data should be encrypted and which encryption algorithms should be used. Encryption and decryption may take a lot of time, therefore their use should be limited whenever possible. The `Information Obscurity` pattern resolves this issue by utilizing a specific confidentiality level to determine what exactly must be concealed with encryption.

Implementation examples

Example of an `Information Obscurity` pattern implementation: Secure Login example.

Sources of information

The `Information Obscurity` pattern is described in detail in the following resources:

- Dangler, Jeremiah Y., "Categorization of Security Design Patterns" (2013). Electronic Theses and Dissertations. Paper 1119. https://dc.etsu.edu/etd/1119
- Schumacher, Markus, Fernandez-Buglioni, Eduardo, Hybertson, Duane, Buschmann, Frank, and Sommerlad, Peter. "Security Patterns: Integrating Security and Systems Engineering" (2006)

Secure Login (Boost, TLS-terminator) example

The `Secure Login` example demonstrates use of the Information Obscurity pattern. This example demonstrates the capability to transmit critical system information through an untrusted environment.

Example architecture

This example simulates the acquisition of remote access to an IoT device by sending user account credentials (user name and password) to this device. The untrusted environment within the IoT device is the web server that responds to requests from users. Practical experience has shown that this kind of web server is easy to detect and frequently attacked successfully because IoT devices do not have built-in tools for protection against intrusion and other attacks. Users also gain access to the IoT device through an untrusted network. Obviously, encryption algorithms must be used in these types of conditions to protect user account credentials from being compromised.

In terms of the architecture in these systems, the following objects can be distinguished:

- Data source: user's browser.
- Point of communication with the device: web server.
- Subsystem for processing information from the user: authentication subsystem.

To employ cryptographic protection, the following steps must be completed:

1. Configure interaction between the data source and the device over the HTTPS protocol. This helps prevent unauthorized surveillance of HTTP traffic and MITM (man-in-the-middle) attacks.

2. Generate a shared secret between the data source and the information processing subsystem.

3. Use this secret to encrypt information on the data source side and to decrypt the information on the information processing subsystem side. This helps prevent data within the device from being compromised (at the point of communication).

The `Secure Login` example includes the following components:

- `Boost` web server (untrusted component, `WebServer` program).

- User authentication subsystem (trusted component, `AuthService` program).

- TLS Terminator (trusted component, `TlsEntity` program). This component supports the TLS (transport layer security) mechanism. Together with the web server, TLS Terminator supports the HTTPS protocol on the device side (the web server interacts with the browser through TLS Terminator).

The user authentication process occurs as follows:

1. Using their browser, the user opens the page at `https://localhost:1106` (when running the example in QEMU) or at `https://<platform IP address>:1106` (when running the example on the target hardware platform). HTTP traffic between the browser and TLS Terminator will be transmitted in encrypted form, but the web server will work only with unencrypted HTTP traffic.

   > This example uses a self-signed certificate, so most up-to-date browsers will warn you that the connection is not secure. You need to agree to use this "insecure" connection, which will actually be encrypted despite the warning. When using certain browsers, the message `"mbedtlsErrorCode=-0x7780, Error performing SSL handshake: SSL - A fatal alert message was received from our peer"` may appear.

2. The `Boost` web server running in the `WebServer` program displays the `index.html` page, which contains an authentication prompt.

3. The user clicks the `Log in` button.

4. The `WebServer` program queries the `AuthService` program via IPC to get the page containing the user name and password input form.

5. The `AuthService` program performs the following actions:

   - Generates a private key and public settings, and calculates the public key based on the Diffie-Hellman algorithm.

   - Creates the `auth.html` page containing the user name and password input form (the page code contains the public settings and the public key).

   - Transfers the received page to the `WebServer` program via IPC.

6. The `Boost` web server running in the `WebServer` program displays the `auth.html` page containing the user name and password input form.

7. The user completes the form and clicks the `Submit` button (correct data for authentication is contained in the file `secure_login/auth_service/src/authservice.cpp`).

8. The `auth.html` page code executed by the browser performs the following actions:

   - Generates a private key and calculates the public key and shared secret key based on the Diffie-Hellman algorithm.
   - Encrypts the password by using the `XOR` operation with the shared secret key.
   - Transmits the user name, encrypted password and public key to the web server.

9. The `WebServer` program queries the `AuthService` program via IPC to get the page containing the authentication result by transmitting the user name, encrypted password and public key.

10. The `AuthService` program performs the following actions:

    - Calculates the shared secret key based on the Diffie-Hellman algorithm.
    - Decrypts the password by using the shared secret key.
    - Returns the `result_err.html` page or `result_ok.html` page depending on the authentication result.

11. The `Boost` web server running in the `WebServer` program displays the `result_err.html` page or the `result_ok.html` page.

This way, confidential data is transmitted only in encrypted form through the network and web server. In addition, all HTTP traffic is transmitted through the network in encrypted form. Data is transferred between components via IPC interactions controlled by the Kaspersky Security Module.

Dynamically created IPC channels

IPC connections between programs are established dynamically using the DCM system program, in which servers publish their interfaces and clients can connect to previously published interfaces. Please note that the same program can act as a client and server at the same time and interact with DCM as both a client and a server.

Unit testing using the GoogleTest framework

In addition to the Information Obscurity pattern, the `Secure Login` example demonstrates use of the GoogleTest framework to conduct unit testing of applications developed for KasperskyOS (this framework is provided in KasperskyOS Community Edition SDK).

The source code of the tests is located at the following path:

`/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples/secure_login/tests`

These unit tests are designed for verification of certain CPP modules of the authentication subsystem and web server.

To start testing:

1. Go to the directory with the `Secure Login` example.
2. Delete the `build` directory containing the results of the previous build by running the following command:

   ```
   sudo rm -rf build/
   ```

3. Run the commands to start testing:

   ```
   $ KL_WITH_TEST=YES
   $ cmake -B build -D CMAKE_TOOLCHAIN_FILE="$KOSCEDIR/toolchain/share/toolchain-*-kos.cmake" && cmake --build
   build --target sim
   ```

Tests are conducted in the `TestEntity` program. The `AuthService` and `WebServer` programs are not started in this case. Therefore, the example cannot be used to demonstrate the Information Obscurity pattern when testing is being conducted.

After testing is finished, the results of the tests are displayed.

Example files

The code of the example and build scripts are available at the following path:

```
/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples/secure_login
```

Building and running the example

The CMake system, which is included with KasperskyOS Community Edition SDK, is used to build and run the example.

See also Building and running examples section.

Examples included in the SDK

In addition to the Developer's Guide, the KasperskyOS Community Edition SDK provides examples of KasperskyOS-based solutions. Examples are located in the `/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples` directory. Each example resides in a separate directory containing the following:

- Directories containing the source code of programs.
- The `resources` directory contains the formal specification of the solution component. This directory may optionally contain files necessary for the example to work. For example, it may contain the network configuration files `hosts`, `dhcpcd.conf` and `ntp.conf`.
- Solution build scenarios in the form of `CMakeLists.txt` files.
- `README.md` files. These files have the same type of contents and refer to this Guide.

Examples of basic solutions

The table below provides a description of examples for developing basic solutions.

Basic solutions

Example directory in the SDK	Brief description of the example	Running in QEMU	Running in Raspberry Pi 4 B	Running in OrangePi RV2
hello	Demonstrates the most basic solution. The `Hello` program sends a message to standard error. For more details, refer to hello example.	Yes	Yes	Yes
echo	Demonstrates interaction between programs via IPC. The `Client` sends data to the `Server` program, receives the modified data in return, and prints that data to standard error. For more details, refer to echo example.	Yes	Yes	Yes
ping	Demonstrates use of a solution security policy to control IPC interaction between programs. The two interface methods `Ping` and `Pong` provided by the `Server` program are called by the `Client` program in a varying sequence. However, the solution security policy allows only alternating calls of `Ping` and `Pong`. For more details, refer to ping example.	Yes	Yes	Yes
hello_from_rust	Demonstrates how to let your solution include a simple program that was developed based on Rust and built using the Cargo build system and package manager. The `Hello` program sends a message to standard error. For more details, refer to hello_from_rust example.	Yes	Yes	Yes
hello_corrosion	Demonstrates how to let your solution include a simple program that was developed based on Rust and uses Corrosion, which is a set of libraries for integrating Rust into CMake projects. The `Hello` program sends a message to standard error. For more details, refer to hello_corrosion example.	Yes	Yes	Yes

Example use of the file systems and network stack

The table below provides a description of examples that demonstrate the use of file systems and a network stack.

Example directory in the SDK	Brief description of the example	Running in QEMU	Running in Raspberry Pi 4 B	Running in OrangePi RV2
embedded_vfs	Demonstrates how to let your solution include a system program that implements file system support. The `Client` program creates a directory, creates and deletes a file, and reads from a file or writes to a file. For more details, refer to embedded_vfs example.	Yes	Yes	Yes
net_with_separate_vfs	Demonstrates how to establish a connection between programs running in KasperskyOS via TCP sockets using a loopback interface. The `Client` program creates a TCP socket, connects to the server through the loopback interface, sends data to the server, and closes the connection. The `Server` program accepts the connection request through the TCP socket, receives data from the client, and then closes the connection. For more details, refer to net_with_separate_vfs example.	Yes	Yes	Yes
net2_with_separate_vfs	Demonstrates how to establish a connection via TCP sockets between the client program running in KasperskyOS and the server program running in the host operating system. The `Client` program creates a TCP socket, connects to the server, sends data to the server, and closes the connection. The `Server` program accepts the connection request through the TCP socket, receives data from the client, and then closes the connection. For more details, refer to net2_with_separate_vfs example.	Yes	Yes	Yes
vfs_extfs	Demonstrates how to mount block device file systems (ext2, ext3, ext4). The `FileVfs` program implements the server portion of the virtual file system, thereby providing the interface used for interaction with file systems and block devices. The `Client` program sends requests to the `FileVfs` program via IPC to mount file systems to the specified directories and perform basic operations with files (create, write, read, and delete) to verify that the file systems are mounted correctly. For more details, refer to vfs_extfs example.	Yes	Yes	Yes
vfs_relay	Demonstrates how to use the VFS relay mechanism to separate data streams across multiple VFS system programs. The Client program creates and uses files located in different directories. The appropriate VFS program for file operations is chosen based on the directory. For more details, refer to vfs_relay example.	Yes	Yes	Yes
multi_vfs_ntpd	Demonstrates support of the Network Time Protocol (NTP). The `Ntpd` program is provided in KasperskyOS Community Edition SDK and is an implementation of an NTP client, which synchronizes the system time with NTP servers. The `Client` program changes the current system time to the time specified in the macro. After the time is synchronized by the `Ntpd` program, the year in the received system time is expected to differ from the year that was previously set by the macro. For more details, refer to multi_vfs_ntpd example.	Yes	Yes	Yes
multi_vfs_dns_client	Demonstrates use of the Domain Name System (DNS) for domain name resolution in KasperskyOS. After checking the network connection, the `Client` program receives information about the DNS server by using its fully qualified domain name and port. Then it resolves the domain name to an IP address and verifies that the IP address matches the previously predefined values. For more details, refer to multi_vfs_dns_client example.	Yes	Yes	Yes
multi_vfs_dhcpcd	Demonstrates support of the network protocol named Dynamic Host Configuration Protocol (DHCP) in KasperskyOS. The `Dhcpcd` program is provided in KasperskyOS Community Edition SDK and is an implementation of a DHCP client, which configures network interfaces. The `Client` program receives information about the configured network interfaces and prints them to standard error. For more details, refer to multi_vfs_dhcpcd example.	Yes	Yes	Yes
mqtt_publisher	Demonstrates support of the Message Queue Telemetry Transport (MQTT) protocol for exchanging messages in KasperskyOS. The `Publisher` program is an implementation of an MQTT publisher running in KasperskyOS that publishes the current time in a loop. The MQTT subscriber running in the host operating system prints messages received from the MQTT publisher to standard output. For more details, refer to mqtt_publisher example.	Yes	Yes	Yes

Example directory in the SDK	Brief description of the example	Running in QEMU	Running in Raspberry Pi 4 B	Running in OrangePi RV2
mqtt_subscriber	Demonstrates support of the Message Queue Telemetry Transport (MQTT) protocol for exchanging messages in KasperskyOS. The `Subscriber` program is an implementation of an MQTT subscriber running in KasperskyOS that prints messages to standard output when it receives those messages from the MQTT publisher running in the host operating system. For more details, refer to mqtt_subscriber example.	Yes	Yes	Yes
virtiofs	This example demonstrates support for the high-performance virtual file system Virtiofs. For more details, refer to the virtiofs example.	Yes	No	No

Driver usage examples

The table below provides a description of examples using the drivers provided in KasperskyOS Community Edition SDK to work with GPIO, I2C, UART, SPI, USB, and PWM hardware interfaces.

Using drivers in KasperskyOS-based solutions

Example directory in the SDK	Brief description of the example	Running in QEMU	Running in Raspberry Pi 4 B	Running in OrangePi RV2
gpio_input	Demonstrates use of the General-Purpose Input/Output (GPIO) driver for input via GPIO pins. The `Client` program configures the GPIO pins to output mode and reads their values. For more details, refer to gpio_input example.	No	Yes	Yes
gpio_output	Demonstrates use of the GPIO driver for output via GPIO pins. The `Client` program configures GPIO pins to output mode and switches them to a logical one (voltage on the pin) and then to a logical zero (no voltage on the pin). For more details, refer to gpio_output example.	No	Yes	Yes
gpio_interrupt	Demonstrates use of the GPIO driver to verify the operation of interrupts for GPIO pins. The `Client` program configures GPIO pins to input mode and registers interrupt handling functions for all GPIO pins except those specified in the list of exclusions. If an interrupt is triggered on a GPIO pin, the `Client` program removes the handler function for this GPIO pin. The program terminates when an interrupt occurs on each GPIO pin. For more details, refer to gpio_interrupt example.	No	Yes	Yes
gpio_echo	Demonstrates use of the GPIO driver to verify the input/output functionality of GPIO pins and the operation of interrupts for GPIO pins. The `Client` program configures two interconnected pins: one pin to input mode and one pin to output mode. It then registers the interrupt handler for the input pin and changes the state of the output pin several times while waiting for the interrupt handler to be triggered. For more details, refer to gpio_echo example.	No	Yes	Yes
i2c_ds1307_rtc	Demonstrates use of the Inter-Integrated Circuit (I2C) driver on hardware platforms. The `I2cClient` program verifies operation of the real-time clock module on the DS1307Z chip via the I2C driver interface by setting and verifying the date on the device. For more details, refer to i2c_ds1307_rtc example.	No	Yes	Yes
uart	Demonstrates use of the Universal Asynchronous Receiver-Transmitter (UART) driver. The `Client` program performs the following operations with UART: initialization, opening a port, sending data, and closing a port. For more details, refer to uart example.	Yes	Yes	Yes
spi_check_regs	Demonstrates use of the Serial Peripheral Interface (SPI) driver. The `Client` program opens an SPI channel, prints its parameters to standard output and sets the required operating mode. The program then uses this SPI channel to forward the data sequence and waits to receive the ID of the ATTiny controller installed on the Sense HAT board. For more details, refer to spi_check_regs example.	No	Yes	Yes

Example directory in the SDK	Brief description of the example	Running in QEMU	Running in Raspberry Pi 4 B	Running in OrangePi RV2
barcode_scanner	Demonstrates use of a Universal Serial Bus (USB) driver via the `libevdev` library. The `BarcodeScanner` program uses the `libevdev` library for interaction with a barcode scanner connected to the hardware platform's USB port. For more details, refer to barcode_scanner example.	No	Yes	Yes
watchdog_system_reset	Demonstrates use of the Watchdog driver for monitoring the state of KasperskyOS and automatically restarting the Raspberry Pi 4 B. The `Client` program interacts with the Watchdog timer as follows: changes the default value of the timer to a new value and starts the timer, resets the timer several times, and waits for the operating system to restart when the timer is triggered. For more details, refer to watchdog_system_reset example.	No	Yes	No
mass_storage	Demonstrates use of the UsbMassStorage driver for working with an external USB drive connected to the hardware platform's USB port. The `FileVfs` program implements the server portion of the virtual file system, thereby providing the interface used for interaction with file systems and block devices. The `Client` program sends requests to the `FileVfs` program via IPC to mount file systems to the specified directories and perform basic operations with files (create, write, read, and delete) to verify that the file systems are mounted correctly on an external USB drive. For more details, refer to mass_storage example.	No	Yes	Yes

Library usage examples

The table below provides examples of using the libraries and components included in KasperskyOS Community Edition SDK to perform various functions, such as logging, applying regular expressions, exchanging messages via IPC, and testing performance.

Using libraries in solutions

Example directory in the SDK	Brief description of the example	Running in QEMU	Running in Raspberry Pi 4 B	Running in OrangePi RV2
shared_libs	Demonstrates the use of dynamic and static libraries in KasperskyOS. The `Client` program calls functions from a static library, a dynamic library linked together with the program, and a dynamic library loaded into memory when the `dlopen()` function of the POSIX interface is called. For more details, refer to shared_libs example.	Yes	Yes	Yes
logrr	Demonstrates how to use the LogRR component to log the operation of programs in KasperskyOS. The `Client` program sends messages to the log server, which saves these messages to a file via an output channel. Messages are sent using various log levels. For more details, refer to logrr example.	Yes	Yes	Yes
pcre	Demonstrates use of the `pcre` library for working with regular expressions in KasperskyOS. The `Client` program searches for all occurrences of the necessary word within the defined text fragment and counts the number of these occurrences. For more details, refer to pcre example.	Yes	Yes	Yes
iperf_separate_vfs	Demonstrates use of the `iperf` library to test network performance. The `Server` program configures the network interface, then creates and runs a network performance test using the `iperf` library. For more details, refer to iperf_separate_vfs example.	Yes	Yes	Yes
pal_tests	Demonstrates use of the Policy Assertion Language (PAL) when writing tests for a KasperskyOS-based solution security policy. These tests check the solution security policy prior to starting development of the software code based on the formal specifications of solution components. For more details, refer to pal_tests example.	Yes	No	No

Example directory in the SDK	Brief description of the example	Running in QEMU	Running in Raspberry Pi 4 B	Running in OrangePi RV2
framebuffer_qr_writer	Demonstrates how to set up a graphical user interface in a KasperskyOS-based solution using a set of cross-platform libraries: SDL2 for managing the application window and processing user data input, Nuklear for building the graphical interface, Skia as a graphics rendering engine, and ZXing for generating QR codes. For more details, refer to framebuffer_qr_writer example.	No	Yes	No
dump_collector	Demonstrates the mechanism for creating a crash dump in a KasperskyOS-based solution. For more details, refer to dump_collector example.	Yes	Yes	Yes
kpa_example	Demonstrates how to work with KPA packages and processes using the PackageManager and ExecutionManager components. For more details, refer to kpa_example.	Yes	Yes	Yes
uart_toybox	Demonstrates how to work with the Terminal component in a KasperskyOS-based solution. For more details, refer to uart_toybox example.	Yes	Yes	Yes
perfcnt	Demonstrates the use of performance counters in a KasperskyOS-based solution. The `Worker` program periodically loads the processor and uses memory. The `Monitor` program gets the performance counter values for the `Worker` program. For more details, refer to perfcnt example.	Yes	Yes	Yes
kcfm_echo_server	Demonstrates integration of the KCFM component into a KasperskyOS-based web server where an application behavior signature is generated, packaged into a KPA package along with an ELF file, and compared with the actual behavior at startup to detect anomalies. The process includes deploying the package, starting the application through KCFM, and monitoring its operation to detect deviations from the specified signature. For more details, refer to the kcfm_echo_server example.	Yes	Yes	No

Security pattern usage examples

The table below shows examples of using security patterns in KasperskyOS.

Using security patterns in solutions

Example directory in the SDK	Brief description of the example	Running in QEMU	Running in Raspberry Pi 4 B	Running in OrangePi RV2
secure_logger	Demonstrates use of the Distrustful Decomposition pattern for separating event log read/write operations. The `Application` program generates a message and forwards it to the `Logger` program via IPC. The `Logger` program receives the message and writes it to the event log residing on the drive. The `Reader` program reads the message from the event log and forwards it to the `LogViewer` program via IPC. The `LogViewer` program prints the message to standard output. For more details, refer to secure_logger example.	Yes	Yes	Yes
separate_storage	Demonstrates use of the Distrustful Decomposition pattern to separate data storage for trusted and untrusted programs. The example contains two programs: `UserManager` and `CertificateManager`. Each of these programs uses its own instance of the VFS program to access a separate file system. Each VFS program instance includes a block device driver linked to an individual logical drive partition. The `UserManager` program does not have access to the file system of the `CertificateManager` program, and vice versa. For more details, refer to separate_storage example.	Yes	Yes	Yes
defer_to_kernel	Demonstrates use of the Defer to Kernel and Policy Decision Point patterns to guarantee isolation of running programs (processes) by the KasperskyOS kernel. In this example, the `ValidPictureClient` and `NonValidPictureClient` programs query the `PictureManager` program to receive information. However, only the `ValidPictureClient` program is allowed to interact with the `PictureManager` program. For more details, refer to defer_to_kernel example.	Yes	Yes	Yes

Example directory in the SDK	Brief description of the example	Running in QEMU	Running in Raspberry Pi 4 B	Running in OrangePi RV2
device_access	Demonstrates use of the Privilege Separation pattern in which different programs are responsible for authorization and access to data. The `Device` program queries the `Storage` program to receive information and queries the `LoginManager` program for authorization. The `Device` program can query the `Storage` program only after successful authorization in the `LoginManager` program. For more details, refer to device_access example.	Yes	Yes	Yes
secure_login	Demonstrates use of the Information Obscurity pattern for the capability to transmit critical system information through an untrusted environment. The `Boost` web server interacts with the browser to process authentication requests, and data is transmitted between the browser and the server via TLS Terminator for traffic encryption. The `AuthService` program generates encryption keys to encrypt passwords based on the Diffie-Hellman algorithm. For more details, refer to secure_login example.	Yes	Yes	Yes

hello example

The example demonstrates the most basic KasperskyOS-based solution. Source code of the `./hello/src/hello.c` application looks simple and familiar to a developer that uses C, and is fully compatible with POSIX.

Example directory in the SDK

The code of the example and build scripts are available at the following path:

```
/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples/hello
```

List of programs

- The example includes system programs supplied in the KasperskyOS boot image and additional system programs and drivers required for working with the emulator or a specific hardware platform on which the solution will run. For more details, refer to Architecture and startup of a KasperskyOS-based solution.

  System programs are added to the solution image automatically during the build.

- `Hello` – application that writes a message to standard error output. The program build script (the `./hello/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

Security policy description

The security policy description file for a KasperskyOS-based solution is located in the `./einit/src/security.psl` directory.

> This policy can be used only as a stub during the early stages of a solution development so that the security module does not interfere with interactions. It would be unacceptable to apply such a policy in a real-world KasperskyOS-based solution.

Additional files

The `./einit/CMakeLists.txt` file contains instructions for building a KasperskyOS boot image.

The `./resources/xdl/hello` directory contains the `Hello.edl` file, which is the EDL description for the `Hello` program.

The `./einit/src/` directory contains the configuration files for the ServiceManager system program, which is responsible for installing and running KPA packages:

- The `initial_install.json` file contains a list of KPA packages that must be installed when KasperskyOS is started for the first time.
- The `system_services.json` file contains a list of installed programs that must be started each time KasperskyOS starts.

Building and running the example

The `CMake` system, which is included with KasperskyOS Community Edition SDK, is used to build and run the example.

See Building and running examples section.

echo example

The echo example demonstrates the use of IPC transport.

It shows how to use the main tools that let you implement interaction between programs.

IPC connections between programs are established dynamically using the DCM system program, in which servers publish their interfaces and clients can connect to previously published interfaces. Please note that the same program can act as a client and server at the same time and interact with DCM as both a client and a server.

Example directory in the SDK

The code of the example and build scripts are available at the following path:

```
/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples/echo
```

List of programs

- The example includes system programs supplied in the KasperskyOS boot image and additional system programs and drivers required for working with the emulator or a specific hardware platform on which the solution will run. For more details, refer to Architecture and startup of a KasperskyOS-based solution.

  System programs are added to the solution image automatically during the build.

- `Client` – application that sends the message. The program build script (the `./client/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

- `Server` – application that receives and processes a message and then sends a response. The program build script (the `./server/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

Security policy description

The security policy description file for a KasperskyOS-based solution is located in the `./einit/src/security.psl` directory.

This policy can be used only as a stub during the early stages of a solution development so that the security module does not interfere with interactions. It would be unacceptable to apply such a policy in a real-world KasperskyOS-based solution.

Additional files

The `./einit/CMakeLists.txt` file contains instructions for building a KasperskyOS boot image.

The `./resources/xdl/echo` directory contains files of formal specifications for programs, components, and interfaces.

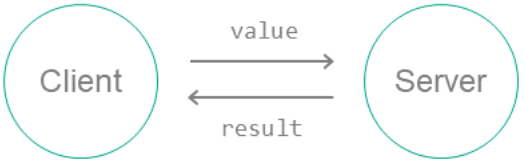
The `./einit/src/` directory contains the configuration files for the ServiceManager system program, which is responsible for installing and running KPA packages:

- The `initial_install.json` file contains a list of KPA packages that must be installed when KasperskyOS is started for the first time.
- The `system_services.json` file contains a list of installed programs that must be started each time KasperskyOS starts.

Operating scenario

The echo example describes a basic case of interaction between two programs:

1. The `Client` program sends a number (`value`) to the `Server` program.
2. The `Server` program modifies this number and sends the new number (`result`) to the `Client` program.
3. The `Client` program prints the `result` number to the screen.

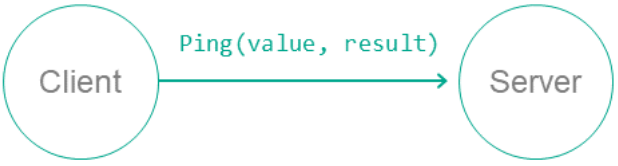


To set up this interaction between programs:

1. Use the DCM system program to connect the `Client` and `Server` programs. The server must publish its interfaces to DCM, and the client must connect to them through DCM.
2. On the server, implement an interface with a single `Ping` method that has one input argument (the original number (`value`)) and one output argument (the modified number (`result`)).

   Description of the `Ping` method in the IDL language:

   ```
   Ping(in UInt32 value, out UInt32 result);
   ```



3. Create static description files in the EDL, CDL, and IDL languages and generate files containing the transport methods and types (proxy object, dispatchers, etc.).

4. In the code of the `Client` program, initialize all required objects (transport, proxy object, request structure, etc.) and call the interface method.

5. In the code of the `Server` program, prepare all the required objects (transport, component dispatcher and program dispatcher, etc.), accept the request from the client, process it and send a response.

Building and running the example

The `CMake` system, which is included with KasperskyOS Community Edition SDK, is used to build and run the example.

See Building and running examples section.

ping example

Demonstrates use of a solution security policy to control IPC interaction between programs.

IPC connections between programs are established dynamically using the DCM system program, in which servers publish their interfaces and clients can connect to previously published interfaces. Please note that the same program can act as a client and server at the same time and interact with DCM as both a client and a server.

Example directory in the SDK

The code of the example and build scripts are available at the following path:

```
/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples/ping
```

List of programs

- The example includes system programs supplied in the KasperskyOS boot image and additional system programs and drivers required for working with the emulator or a specific hardware platform on which the solution will run. For more details, refer to Architecture and startup of a KasperskyOS-based solution.

  System programs are added to the solution image automatically during the build.

- `Client` is an application that calls the `Ping()` and `Pong()` methods. The program build script (the `./client/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

- `Server` is an application that implements the `Ping()` and `Pong()` methods, which increase the number received from the client by a fixed amount and return the result. The program build script (the `./server/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

Security policy description

The `security.psl` file contains description of the solution security policy.

The `core.psl` file, which contains part of the security policy description regulating the interactions between programs and the KasperskyOS kernel, is included in the `security.psl` file via the `use` declaration.

The declarations in the PSL files are provided with comments that explain the purpose of these declarations.

The solution security policy in this example allows startup of the KasperskyOS kernel and system programs. Queries to the `Server` program are managed by methods of the Flow security model.

The finite-state machine described in the configuration of the `request_state` Flow security model object has two states: `not_sent` and `sent`. The initial state is `not_sent`. Only transitions from `not_sent` to `sent` and vice versa are allowed.

When the `Ping()` and `Pong()` methods are called, the current state of the `request_state` object is checked. In the `not_sent` state, only a `Ping()` call is allowed, in which case the state changes to `sent`. Likewise, in the `sent` state, only a `Pong()` call is allowed, in which case the state changes to `not_sent`.

Therefore, the `Ping()` and `Pong()` methods can be called only in succession.

Additional files

The `./resources/xdl/ping` directory contains the following files:

- `Client.edl` and `Server.edl` are EDL descriptions for the example programs.
- `Control.cdl` is the CDL description of the `ping.Control` component.
- `Connection.idl` is an IDL description of the `ping.Connection` package.

The `./einit/CMakeLists.txt` file contains instructions for building a KasperskyOS boot image.

The `./einit/src/` directory contains the configuration files for the ServiceManager system program, which is responsible for installing and running KPA packages:

- The `initial_install.json` file contains a list of KPA packages that must be installed when KasperskyOS is started for the first time.
- The `system_services.json` file contains a list of installed programs that must be started each time KasperskyOS starts.

Operating scenario

1. The `Client` program establishes a connection to the `Server` program and attempts to call the `Ping()` and `Pong()` methods in the following sequence: alternate between `Ping()` and `Pong()` five times, then `Ping()` twice and `Pong()` twice.

2. When the `Ping()` and `Pong()` methods are called, the current state of the `request_state` object tracking the state of the finite-state machine is checked.

   - `Ping()` is allowed only in the `not_sent` state.
   - `Pong()` is allowed only in the `sent` state.
   - The state of the finite-state machine changes after each successful call.

3. As a result, the first 10 alternating calls are successful. Subsequent violations of the order (the second consecutive `Ping()` and the second consecutive `Pong()`) are blocked by the security policy.

Building and running the example

The CMake system, which is included with KasperskyOS Community Edition SDK, is used to build and run the example.

See Building and running examples section.

net_with_separate_vfs example

This example demonstrates how to establish a connection between programs running in KasperskyOS via TCP sockets using a loopback interface.

IPC connections between programs are established dynamically using the DCM system program, in which servers publish their interfaces and clients can connect to previously published interfaces. Please note that the same program can act as a client and server at the same time and interact with DCM as both a client and a server.

Example directory in the SDK

The code of the example and build scripts are available at the following path:

```
/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples/net_with_separate_vfs
```

List of programs

- The example includes system programs supplied in the KasperskyOS boot image and additional system programs and drivers required for working with the emulator or a specific hardware platform on which the solution will run. For more details, refer to Architecture and startup of a KasperskyOS-based solution.

  System programs are added to the solution image automatically during the build.

- The `Client` – application that creates a TCP socket, connects to the server through the loopback interface, sends data to the server, and closes the connection. The program build script (the `./client/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

- `Server` – application that accepts the connection request through the TCP socket, receives data from the client, and then closes the connection. The program build script (the `./server/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

- `Launcher` – application that waits for the network interfaces to initialize and then starts the `Server` and `Client` programs. The program build script (the `./launcher/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

Security policy description

The `security.psl` file contains a description of a security policy for a KasperskyOS-based solution and is located in the `./einit/src` directory. The declarations in the PSL file are provided with comments that explain the purpose of these declarations.

Additional files

The `./resources/xdl/net_with_separate_vfs` directory contains the `Client.edl` and `Server.edl` files, which are the EDL descriptions of the `Client` and `Server` programs, respectively.

The `./resources/hdd/etc` directory contains the `hosts` configuration file for the `VfsNet` program.

The `./einit/CMakeLists.txt` file contains instructions for building a KasperskyOS boot image.

The `./einit/src/` directory contains the configuration files for the ServiceManager system program, which is responsible for installing and running KPA packages:

- The `initial_install.json` file contains a list of KPA packages that must be installed when KasperskyOS is started for the first time.

- The `system_services.json` file contains a list of installed programs that must be started each time KasperskyOS starts.

The `./einit/src/product_init.yaml` file contains the startup parameters for the `VfsNet` and `DNetSrv` system programs in Init description format.

Operating scenario

The `Launcher` program waits for the network interfaces to initialize and then starts the `Server` and `Client` programs. The `Client` and `Server` programs establish a TCP connection through the loopback interface. To do so, they use one instance of the network stack. In other words, they interact with a "shared" VFS program (in the example, this is the `VfsNet` program). Both programs use standard POSIX functions in their code. The `Server` program waits for incoming data using the `recv()` function, and the `Client` program sends data over an established TCP connection using the `send()` function.

Building and running the example

The `CMake` system, which is included with KasperskyOS Community Edition SDK, is used to build and run the example.

See Building and running examples section.

net2_with_separate_vfs example

This example demonstrates how to establish a connection via TCP sockets between the client program running in KasperskyOS and the server program running in the host operating system.

Example directory in the SDK

The code of the example and build scripts are available at the following path:

```
/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples/net2_with_separate_vfs
```

List of programs

- The example includes system programs supplied in the KasperskyOS boot image and additional system programs and drivers required for working with the emulator or a specific hardware platform on which the solution will run. For more details, refer to Architecture and startup of a KasperskyOS-based solution.

  System programs are added to the solution image automatically during the build.

- The `Client` – application that creates a TCP socket, connects to the server through the loopback interface, sends data to the server, and closes the connection. The program build script (the `./client/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

- `Server` – application that accepts the connection request through the TCP socket, receives data from the client, and then closes the connection.

Security policy description

The `security.psl` file contains a description of a security policy for a KasperskyOS-based solution and is located in the `./einit/src` directory. The declarations in the PSL file are provided with comments that explain the purpose of these declarations.

Additional files

The `./resources/xdl/net2_with_separate_vfs` directory contains the `Client.edl` file, which is the EDL description of the `Client` program.

The `./einit/CMakeLists.txt` file contains instructions for building a KasperskyOS boot image.

The `./einit/src/` directory contains the configuration files for the ServiceManager system program, which is responsible for installing and running KPA packages:

- The `initial_install.json` file contains a list of KPA packages that must be installed when KasperskyOS is started for the first time.
- The `system_services.json` file contains a list of installed programs that must be started each time KasperskyOS starts.

The `./einit/src/product_init.yaml` file contains the startup parameters for the `VfsNet` and `DNetSrv` system programs in Init description format.

Operating scenario

The `net2_with_separate_vfs` example is a modified `net with separate vfs` example. In contrast to the `net_with_separate_vfs` example, in this example a program interacts over the network with an external server rather than another program running in KasperskyOS.

This example consists of the `Client` program running in KasperskyOS and the `Server` program running in a Linux host operating system. The `Client` program and `Server` program are bound by a TCP socket. Standard POSIX functions are used in the code of the `Client` program.

To connect the `Client` program and the `Server` program using a socket, the `Client` program must interact with the `VfsNet` program. During the build, the `VfsNet` program is linked to a network driver that supports interaction with the `Server` program running in Linux.

The IP address of the `Server` program is defined in the `./CMakeList.txt` file by using the `SERVER_IP` variable and is set to `10.0.2.2` by default. The default values for the `Client` program (interface name, address, network mask, and gateway address) are taken from the file `/opt/KasperskyOS-Community-Edition-<platform>-<version>/sysroot-*-kos/include/kos_net.h`. You can change these values according to the configuration of your network in the `client/src/client.c` file located in the example directory.

Building and running the example

The `CMake` system, which is included with KasperskyOS Community Edition SDK, is used to build and run the example.

To ensure that an example runs correctly, you must run the `Server` program in a Linux host operating system or on a computer connected to the hardware platform.

If the example is run in QEMU, the `Server` program will be built and run in the Linux host operating system automatically. After performing the build, the `server` executable file of the `Server` program is located in the following directory:

```
<example nam>/build/host/server/
```

If the example is run on a hardware platform, you must independently build the executable file of the `Server` program by running the following commands:

```
$ export PATH="/opt/KasperskyOS-Community-Edition-<platform>-<version>/toolchain/bin/:$PATH"
$ cd net2_with_separate_vfs/server/
$ cmake -G "Unix Makefiles" -B ./build
$ cmake --build ./build/ --target server
```

See Building and running examples section.

embedded_vfs example

This example demonstrates how to include the virtual file system (VFS) provided in KasperskyOS Community Edition SDK into KasperskyOS-based solution being developed.

Example directory in the SDK

The code of the example and build scripts are available at the following path:

```
/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples/embedded_vfs
```

List of programs

- The example includes system programs supplied in the KasperskyOS boot image and additional system programs and drivers required for working with the emulator or a specific hardware platform on which the solution will run. For more details, refer to Architecture and startup of a KasperskyOS-based solution.

  System programs are added to the solution image automatically during the build.

- `Client` – application that fully encapsulates the VFS implementation from KasperskyOS Community Edition SDK. This lets you eliminate the use of IPC for all the standard I/O functions (`stdio.h`, `socket.h`, etc.) for debugging or performance improvement purposes, for example. The program build script (the `./client/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

Security policy description

The `security.psl` file contains a description of a security policy for a KasperskyOS-based solution and is located in the `./einit/src` directory. The declarations in the PSL file are provided with comments that explain the purpose of these declarations.

The solution security policy description in this example allows any interaction between different processes and between processes and the kernel. This policy can be used only as a stub during the early stages of a solution development so that the security module does not interfere with interactions.

Additional files

The directory `./resources/xdl/embedded_vfs` contains the `Client.edl` file, which is the EDL description for the `Client` program.

The `./einit/CMakeLists.txt` file contains instructions for building a KasperskyOS boot image.

The `./einit/src/` directory contains the configuration files for the ServiceManager system program, which is responsible for installing and running KPA packages:

- The `initial_install.json` file contains a list of KPA packages that must be installed when KasperskyOS is started for the first time.

- The `system_services.json` file contains a list of installed programs that must be started each time KasperskyOS starts.

Operating scenario

The `Client` program tests the following operations:

- Create a folder.
- Create and delete a file.
- Read from a file and write to a file.

Building and running the example

The `CMake` system, which is included with KasperskyOS Community Edition SDK, is used to build and run the example.

See Building and running examples section.

vfs_extfs example

This example demonstrates how to mount file systems of a block device.

Example directory in the SDK

The code of the example and build scripts are available at the following path:

```
/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples/vfs_extfs
```

List of programs

- The example includes system programs supplied in the KasperskyOS boot image and additional system programs and drivers required for working with the emulator or a specific hardware platform on which the solution will run. For more details, refer to Architecture and startup of a KasperskyOS-based solution.

  System programs are added to the solution image automatically during the build.

- `Client` sends requests via IPC to the `FileVfs` program. The program build script (the `./client/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

- `FileVfs` implements the server portion of the virtual file system, thereby providing the interface used for interaction with file systems and block devices.

Security policy description

The `security.psl` file contains a description of a security policy for a KasperskyOS-based solution and is located in the `./einit/src` directory. The declarations in the PSL file are provided with comments that explain the purpose of these declarations.

Additional files

The directory `./resources/xdl/vfs_extfs` contains the `Client.edl` and `FileVfs.edl` files, which are the EDL descriptions for the example programs.

The `./einit/CMakeLists.txt` file contains instructions for building a KasperskyOS boot image.

The `./einit/src/` directory contains the configuration files for the ServiceManager system program, which is responsible for installing and running KPA packages:

- The `initial_install.json` file contains a list of KPA packages that must be installed when KasperskyOS is started for the first time.
- The `system_services.json` file contains a list of installed programs that must be started each time KasperskyOS starts.

The `./einit/src/product_init.yaml` file contains the startup parameters for the `FileVfs` program in Init description format.

Operating scenario

The `Client` program interacts with the `FileVfs` program, which acts as a VFS server through an IPC interface. The `FileVfs` program mounts file systems (`ext2`, `ext3`, `ext4`) to the specified directories and perform basic operations with files (create, write, read, and delete) to verify that the file systems are mounted correctly.

Building and running the example

The CMake system, which is included with KasperskyOS Community Edition SDK, is used to build and run the example.

See Building and running examples section.

multi_vfs_ntpd example

This example demonstrates support for the Network Time Protocol (NTP) to synchronize the system time in KasperskyOS, and integration of multiple virtual file systems (hereinafter referred to as "VFS") in one solution. For more information about using multiple VFS programs, please refer to the File systems and network section.

Example directory in the SDK

The code of the example and build scripts are available at the following path:

```
/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples/multi_vfs_ntpd
```

List of programs

- The example includes system programs supplied in the KasperskyOS boot image and additional system programs and drivers required for working with the emulator or a specific hardware platform on which the solution will run. For more details, refer to Architecture and startup of a KasperskyOS-based solution.

  System programs are added to the solution image automatically during the build.

- `Client` – application that changes the current system time. The program build script (the `./client/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

- `Launcher` – application that waits for the network interfaces to initialize and then starts the `Client` program. The program build script (the `./launcher/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

Security policy description

The `security.psl` file contains a description of a security policy for a KasperskyOS-based solution and is located in the `./einit/src` directory. The declarations in the PSL file are provided with comments that explain the purpose of these declarations.

Additional files

The `./resources/xdl/multi_vfs_ntpd` directory contains the `Client.edl` file, which provides the formal specification of the KasperskyOS-based solution component.

The `./resources/hdd/etc` directory contains the configuration files for the `VfsNet`, `Dhcpcd` and `Ntpd` programs: `hosts`, `dhcpcd.conf` and `ntp.conf`, respectively.

The `./resources/hdd/var/db/dhcpcd` directory is used by the `Dhcpcd` program to store network connection data.

The `./einit/CMakeLists.txt` file contains instructions for building a KasperskyOS boot image.

The `./einit/src/` directory contains the configuration files for the ServiceManager system program, which is responsible for installing and running KPA packages:

- The `initial_install.json` file contains a list of KPA packages that must be installed when KasperskyOS is started for the first time.
- The `system_services.json` file contains a list of installed programs that must be started each time KasperskyOS starts.

The `./einit/src/product_init.yaml` file contains the startup parameters for the `VfsNet` and `DNetSrv` system programs in Init description format.

Operating scenario

The `Client` program changes the current system time to the time specified in the macro. After the time is synchronized by the `Ntpd` program, the year in the received system time is expected to differ from the year that was previously set by the macro.

Building and running the example

See Building and running examples section.

multi_vfs_dns_client example

Demonstrates use of the Domain Name System (DNS) resolution system in KasperskyOS, and the integration of several virtual file systems (hereinafter VFS) in one solution. For more information about using multiple VFS programs, please refer to the File systems and network section.

The example uses the VFS relay mechanism to allow the Dhcpcd system program to access configuration files in the file system.

Example directory in the SDK

The code of the example and build scripts are available at the following path:

```
/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples/multi_vfs_dns_client
```

List of programs

- The example includes system programs supplied in the KasperskyOS boot image and additional system programs and drivers required for working with the emulator or a specific hardware platform on which the solution will run. For more details, refer to Architecture and startup of a KasperskyOS-based solution.

  System programs are added to the solution image automatically during the build.

- `Client` – application that uses the functions of the `libc` library to access a DNS server. The program build script (the `./client/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

- `Launcher` – application that waits for the network interfaces to initialize and then starts the `Client` program. The program build script (the `./launcher/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

Security policy description

The `security.psl` file contains a description of a security policy for a KasperskyOS-based solution and is located in the `./einit/src` directory. The declarations in the PSL file are provided with comments that explain the purpose of these declarations.

The security policy file shows how you can restrict access to the file system for specific processes via the VFS relay mechanism. The example security policy uses objects of the following security models: regex, HashSet, flow.

Additional files

The `./resources/xdl/multi_vfs_dns_client` directory contains the `Client.edl` file, which is the EDL description for the `Client` program.

The `./resources/hdd/etc` directory contains the `hosts` and `dhcpcd.conf` configuration files for the `VfsNet` and `Dhcpcd` programs, respectively, and the `resolv.conf` file containing data on the DNS server.

The `./resources/hdd/var/db/dhcpcd` directory is used by the `Dhcpcd` program to store network connection data.

The `./einit/CMakeLists.txt` file contains instructions for building a KasperskyOS boot image.

The `./einit/src/` directory contains the configuration files for the ServiceManager system program, which is responsible for installing and running KPA packages:

- The `initial_install.json` file contains a list of KPA packages that must be installed when KasperskyOS is started for the first time.

- The `system_services.json` file contains a list of installed programs that must be started each time KasperskyOS starts.

The `./einit/src/product_init.yaml` file contains the startup parameters for the `VfsNet` and `DNetSrv` system programs in Init description format.

Operating scenario

The `Client` program uses standard `libc` library functions for contacting an external DNS service. These functions are converted into queries to the `VfsNet` program via IPC.

After checking the network connection, the `Client` program obtains the IP address of the host defined in the root CMakeLists.txt file by using the DNS server specified in the `./resources/hdd/etc/resolv.conf` file. Then the program resolves the domain name to an IP address and verifies that the IP address matches the previously predefined values.

Building and running the example

The `CMake` system, which is included with KasperskyOS Community Edition SDK, is used to build and run the example.

See Building and running examples section.

multi_vfs_dhcpcd example

This example demonstrates support of the network protocol named Dynamic Host Configuration Protocol (DHCP) in KasperskyOS. The `Dhcpcd` program is provided in KasperskyOS Community Edition SDK and is an implementation of a DHCP client, which configures network interfaces.

In addition, the example also demonstrates integration of multiple virtual file systems (VFSes) in a single solution. For more information about using multiple VFS programs, please refer to the File systems and network section.

Example directory in the SDK

The code of the example and build scripts are available at the following path:

```
/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples/multi_vfs_dhcpcd
```

List of programs

- The example includes system programs supplied in the KasperskyOS boot image and additional system programs and drivers required for working with the emulator or a specific hardware platform on which the solution will run. For more details, refer to Architecture and startup of a KasperskyOS-based solution.

  System programs are added to the solution image automatically during the build.

- `Client` – application that uses `libc` library functions for getting information on network interfaces (`ioctl`). These functions are converted into queries to the VFS program via IPC. The program build script (the `./client/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

- `Launcher` – application that waits for the network interfaces to initialize and then starts the `Client` program. The program build script (the `./launcher/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

Security policy description

The `security.psl` file contains a description of a security policy for a KasperskyOS-based solution and is located in the `./einit/src` directory. The declarations in the PSL file are provided with comments that explain the purpose of these declarations.

Additional files

The `./resources/xdl/multi_vfs_dhcpcd` directory contains the `Client.edl` file, which is the EDL description for the `Client` program.

The `./einit/CMakeLists.txt` file contains instructions for building a KasperskyOS boot image.

The `./einit/src/` directory contains the configuration files for the ServiceManager system program, which is responsible for installing and running KPA packages:

- The `initial_install.json` file contains a list of KPA packages that must be installed when KasperskyOS is started for the first time.

- The `system_services.json` file contains a list of installed programs that must be started each time KasperskyOS starts.

The `./einit/src/product_init.yaml` file contains the startup parameters for the `VfsNet` and `DNetSrv` system programs in Init description format.

The `./resources/hdd/etc` directory contains:

- `hosts` – configuration file for the `VfsNet` program.
- `dhcpcd.conf`, `dhcpcd_static.conf`, and `dhcpcd_dynamic.conf` are configuration files for the `Dhcpcd` program. The `CMakeLists.txt` root file defines the value of the `DHCPCD_CONFIG` variable that determines the selected configuration file:

  - `FALLBACK`

    Dynamically receive the parameters of network interfaces from an external DHCP server but statically define the parameters if the DHCP server is not available. This value is used by default.

  - `DYNAMIC`

    Dynamically receive the parameters of network interfaces from an external DHCP server.

  - `STATIC`

    Statically define the parameters of network interfaces.

  You can change the default value of the `DHCPCD_CONFIG` variable as follows:

  - In the `CMakeLists.txt` file. For example, you can change it to the `DYNAMIC` value:

    ```
    set (DHCPCD_CONFIG "DYNAMIC" CACHE STRING "dhcpcd config type")
    ```

  - In the terminal when building with `CMake` options:

    ```
    cmake -B build -DDHCPCD_CONFIG=DYNAMIC -D CMAKE_TOOLCHAIN_FILE="$KOSCEDIR/toolchain/share/toolchain-*-kos.cmake" && cmake --build build --target sim
    ```

The `./resources/hdd/var/db/dhcpcd` directory is used by the `Dhcpcd` program to store network connection data.

Operating scenario

The `Dhcpcd` program is provided in KasperskyOS Community Edition SDK and is an implementation of a DHCP client, which configures network interfaces. The `Client` program receives information about the configured network interfaces and prints them to standard error.

Building and running the example

The `CMake` system, which is included with KasperskyOS Community Edition SDK, is used to build and run the example.

See Building and running examples section.

mqtt_publisher (Mosquitto) example

The example demonstrates support for the MQTT (Message Queue Telemetry Transport) messaging protocol in KasperskyOS, and the integration of several virtual file systems (hereinafter VFS) in one solution. For more information about using multiple VFS programs, please refer to the File systems and network section.

Example directory in the SDK

The code of the example and build scripts are available at the following path:

```
/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples/mqtt_publisher
```

List of programs

- The example includes system programs supplied in the KasperskyOS boot image and additional system programs and drivers required for working with the emulator or a specific hardware platform on which the solution will run. For more details, refer to Architecture and startup of a KasperskyOS-based solution.

  System programs are added to the solution image automatically during the build.

- `Publisher` – MQTT publisher application that publishes the current time every 5 seconds. The program build script (the `./publisher/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

- `Launcher` – application that waits for the network interfaces to initialize and then starts the `Publisher` program. The program build script (the `./launcher/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

Security policy description

The `security.psl` file contains a description of a security policy for a KasperskyOS-based solution and is located in the `./einit/src` directory. The declarations in the PSL file are provided with comments that explain the purpose of these declarations.

Additional files

The `./resources/xdl/mqtt_publisher` directory contains the `Publisher.edl` file, which is the EDL description for the `Publisher` program.

The `./resources/hdd/etc` directory contains the configuration files for the `VfsNet`, `Dhcpcd` and `Ntpd` programs: `hosts`, `dhcpcd.conf` and `ntp.conf`, respectively.

The `./resources/hdd/var/db/dhcpcd` directory is used by the `Dhcpcd` program to store network connection data.

The `./einit/CMakeLists.txt` file contains instructions for building a KasperskyOS boot image.

The `./einit/src/` directory contains the configuration files for the ServiceManager system program, which is responsible for installing and running KPA packages:

- The `initial_install.json` file contains a list of KPA packages that must be installed when KasperskyOS is started for the first time.
- The `system_services.json` file contains a list of installed programs that must be started each time KasperskyOS starts.

The `./einit/src/product_init.yaml` file contains the startup parameters for the `VfsNet` and `DNetSrv` system programs in Init description format.

Operating scenario

In this example, an MQTT subscriber must be started on the host operating system, and an MQTT publisher must be started on KasperskyOS. The `Publisher` program, which operates as an MQTT publisher, periodically (at an interval of 5 seconds) publishes the current time in the `"datetime"` topic.

When the example starts and runs successfully, an MQTT subscriber started on the host operating system prints a `"received PUBLISH"` message specifying the `"datetime"` topic and obtained time value.

Building and running the example

The `CMake` system, which is included with KasperskyOS Community Edition SDK, is used to build and run the example.

To run this example, a Mosquitto MQTT broker must be installed and started on the host system. To install and start Mosquitto, run the following commands:

```
$ sudo apt install mosquitto mosquitto-clients
$ sudo /etc/init.d/mosquitto start
```

To start an MQTT subscriber on the host system, run the following command:

```
$ mosquitto_sub -d -t "datetime"
```

See Building and running examples section.

mqtt_subscriber (Mosquitto) example

The example demonstrates support for the MQTT (Message Queue Telemetry Transport) messaging protocol in KasperskyOS, and the integration of several virtual file systems (hereinafter VFS) in one solution. For more information about using multiple VFS programs, please refer to the File systems and network section.

Example directory in the SDK

The code of the example and build scripts are available at the following path:

```
/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples/mqtt_subscriber
```

List of programs

- The example includes system programs supplied in the KasperskyOS boot image and additional system programs and drivers required for working with the emulator or a specific hardware platform on which the solution will run. For more details, refer to Architecture and startup of a KasperskyOS-based solution.

  System programs are added to the solution image automatically during the build.

- `Subscriber` – MQTT subscriber application. The program build script (the `./subscriber/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

- `Launcher` – application that waits for the network interfaces to initialize and then starts the `Subscriber` program. The program build script (the `./launcher/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

Security policy description

The `security.psl` file contains a description of a security policy for a KasperskyOS-based solution and is located in the `./einit/src` directory. The declarations in the PSL file are provided with comments that explain the purpose of these declarations.

Additional files

The `./resources/xdl/mqtt_subscriber` directory contains the `Subscriber.edl` file, which is the EDL description for the `Subscriber` program.

The `./resources/hdd/etc` directory contains the configuration files for the `VfsNet`, `Dhcpcd` and `Ntpd` programs: `hosts`, `dhcpcd.conf` and `ntp.conf`, respectively.

The `./resources/hdd/var/db/dhcpcd` directory is used by the `Dhcpcd` program to store network connection data.

The `./einit/CMakeLists.txt` file contains instructions for building a KasperskyOS boot image.

The `./einit/src/` directory contains the configuration files for the ServiceManager system program, which is responsible for installing and running KPA packages:

- The `initial_install.json` file contains a list of KPA packages that must be installed when KasperskyOS is started for the first time.
- The `system_services.json` file contains a list of installed programs that must be started each time KasperskyOS starts.

The `./einit/src/product_init.yaml` file contains the startup parameters for the `VfsNet` and `DNetSrv` system programs in Init description format.

Operating scenario

In this example, an MQTT publisher must be started on the host operating system, and an MQTT subscriber must be started on KasperskyOS. The `Subscriber` program is an implementation of an MQTT subscriber.

When the example starts and runs successfully, an MQTT subscriber started on KasperskyOS prints a `"Got message with topic: my/awesome/topic, payload: hello"` message.

Building and running the example

The `CMake` system, which is included with KasperskyOS Community Edition SDK, is used to build and run the example.

To run this example, a Mosquitto MQTT broker must be installed and started on the host system. To install and start Mosquitto, run the following commands:

```
$ sudo apt install mosquitto mosquitto-clients
$ sudo /etc/init.d/mosquitto start
```

To start an MQTT publisher on the host system, run the following command:

```
$ mosquitto_pub  -t "my/awesome/topic" -m "hello"
```

See Building and running examples section.

gpio_input example

This example demonstrates use of the GPIO driver for input via GPIO pins.

Example directory in the SDK

The code of the example and build scripts are available at the following path:

```
/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples/gpio_input
```

List of programs

- The example includes system programs supplied in the KasperskyOS boot image and additional system programs and drivers required for working with the emulator or a specific hardware platform on which the solution will run. For more details, refer to Architecture and startup of a KasperskyOS-based solution.

  System programs are added to the solution image automatically during the build.

- `Client` – application configuring the GPIO pins to output mode and reading their values. The program build script (the `./client/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

Security policy description

The `security.psl` file contains a description of a security policy for a KasperskyOS-based solution and is located in the `./einit/src` directory. The declarations in the PSL file are provided with comments that explain the purpose of these declarations.

Additional files

The `./resources/xdl/gpio_input` directory contains the `Client.edl` file, which is the EDL description of the `Client` program.

The `./einit/CMakeLists.txt` file contains instructions for building a KasperskyOS boot image.

The `./einit/src/` directory contains the configuration files for the ServiceManager system program, which is responsible for installing and running KPA packages:

- The `initial_install.json` file contains a list of KPA packages that must be installed when KasperskyOS is started for the first time.

- The `system_services.json` file contains a list of installed programs that must be started each time KasperskyOS starts.

Operating scenario

The `Client` program performs the following actions:

- Initializes drivers: BSP and GPIO.
- Opens GPIO ports named `gpio0`, `gpio1`, ... (the total number of open ports is defined in the `GPIO_BANK_COUNT` macro).
- Configures all GPIO pins except those specified in the `ExceptionPinArr` array as input pins with enabled internal pull-up to a logical zero (`GPIO_DIR_IN | GPIO_PULL_MODE_DOWN`).

  The total number of GPIO pins on the card is defined by the `GPIO_PIN_NUM` macro.

  `ExceptionPinArr` – array of GPIO pin numbers that must be excluded from processing (for example, when used for a debug UART connection).
- Sequentially reads the states of all GPIO pins (except for the excluded ones) and prints messages about these states to the console. The delay between the readings of adjacent pins is defined by the `DELAY_S` macro (the time is indicated in seconds).
- Closes the GPIO ports, deinitializes drivers, and terminates with the `EXIT_SUCCESS` code if there are no errors.

Building and running the example

The `CMake` system, which is included with KasperskyOS Community Edition SDK, is used to build and run the example.

For the `gpio_input` example to work correctly, you must supply 3.3 V (logical 1) to all GPIO pins except those specified in the `ExceptionPinArr` array.

See Building and running examples section.

gpio_output example

This example demonstrates use of the GPIO driver for output via GPIO pins.

Example directory in the SDK

The code of the example and build scripts are available at the following path:

```
/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples/gpio_output
```

List of programs

- The example includes system programs supplied in the KasperskyOS boot image and additional system programs and drivers required for working with the emulator or a specific hardware platform on which the solution will run. For more details, refer to Architecture and startup of a KasperskyOS-based solution.

  System programs are added to the solution image automatically during the build.

- `Client` – application that interacts with the GPIO driver. The program build script (the `./client/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

Security policy description

The `security.psl` file contains a description of a security policy for a KasperskyOS-based solution and is located in the `./einit/src` directory. The declarations in the PSL file are provided with comments that explain the purpose of these declarations.

Additional files

The `./resources/xdl/gpio_output` directory contains the `Client.edl` file, which is the EDL description of the `Client` program.

The `./einit/CMakeLists.txt` file contains instructions for building a KasperskyOS boot image.

The `./einit/src/` directory contains the configuration files for the ServiceManager system program, which is responsible for installing and running KPA packages:

- The `initial_install.json` file contains a list of KPA packages that must be installed when KasperskyOS is started for the first time.

- The `system_services.json` file contains a list of installed programs that must be started each time KasperskyOS starts.

Operating scenario

The example checks the functionality of the GPIO pin output. The total number of GPIO pins on the card is defined by the `GPIO_PIN_NUM` macro. All GPIO pins other than those indicated in the `ExceptionPinArr` array are configured for output. Each GPIO pin will be sequentially changed to a logical one (voltage on the pin) and then to a logical zero. The delay between the changes of GPIO pin state is defined by the `DELAY_S` macro (the time is indicated in seconds).

`ExceptionPinArr` is an array of GPIO pin numbers that need to be excluded from the example. This may be necessary if some GPIO pins are already being used for other functions, e.g. if pins are being used for a UART connection during debugging.

Building and running the example

The `CMake` system, which is included with KasperskyOS Community Edition SDK, is used to build and run the example.

[Optional] Before starting the example, it is recommended to assemble a circuit in which a LED is connected to each GPIO pin (except for those specified in the `ExceptionPinArr` array) through a current-limiting resistor. While the example is running, the LED will light up in the logical 1 state on the corresponding GPIO pin and will go out in the logical zero state. This will let you visually track the change in the state of each GPIO pin.

See Building and running examples section.

gpio_interrupt example

This example demonstrates use of the GPIO driver to verify the operation of interrupts for GPIO pins.

Example directory in the SDK

The code of the example and build scripts are available at the following path:

```
/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples/gpio_interrupt
```

List of programs

- The example includes system programs supplied in the KasperskyOS boot image and additional system programs and drivers required for working with the emulator or a specific hardware platform on which the solution will run. For more details, refer to Architecture and startup of a KasperskyOS-based solution.

  System programs are added to the solution image automatically during the build.

- `Client` – application that interacts with the GPIO driver. The program build script (the `./client/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

Security policy description

The `security.psl` file contains a description of a security policy for a KasperskyOS-based solution and is located in the `./einit/src` directory. The declarations in the PSL file are provided with comments that explain the purpose of these declarations.

Additional files

The `./resources/xdl/gpio_interrupt` directory contains the `Client.edl` file, which is the EDL description of the `Client` program.

The `./einit/CMakeLists.txt` file contains instructions for building a KasperskyOS boot image.

The `./einit/src/` directory contains the configuration files for the ServiceManager system program, which is responsible for installing and running KPA packages:

- The `initial_install.json` file contains a list of KPA packages that must be installed when KasperskyOS is started for the first time.
- The `system_services.json` file contains a list of installed programs that must be started each time KasperskyOS starts.

Operating scenario

The example checks the functionality of interrupts for GPIO pins. The total number of pins on the card is defined by the `GPIO_PIN_NUM` macro. In the `pinsBitmap` bitmask, pins from the `ExceptionPinArr` array are marked as handled so that the example can properly terminate later. All pins other than those indicated in the `ExceptionPinArr` array are switched to the `PINS_MODE` state. An interrupt handler function will be registered for all pins other than those indicated in the `ExceptionPinArr` array.

In an endless loop, the example checks whether the `pinsBitmap` bitmask is equal to the `DONE_BITMASK` bitmask (which corresponds to the condition when an interrupt has occurred on each GPIO pin). The interrupt handler function is then called in this loop. If an interrupt occurs for the first time on a pin that is not specified in the `ExceptionPinArr` array, the interrupt handler function uses the `GpioGetEvent()` function to receive this event. The corresponding pin in the `pinsBitmap` bitmask is marked as handled. Interrupt detection on this pin is disabled by calling the `GpioReleaseMode()` function.

Keep in mind how the example may be affected by the initial state of the registers of pull-up resistors for each pin.

Some GPIO pins may also include external pull-up resistors (for example, they are on pins GPIO2 and GPIO3 in revisions 1.4 and 1.5 of the Raspberry Pi 4 B platform). If these types of resistors are present, weaker internal resistors will not allow these pins to be pulled to logical 0.

Interrupts for the `GPIO_EVENT_LOW_LEVEL` and `GPIO_EVENT_HIGH_LEVEL` events are not supported.

`ExceptionPinArr` is an array of GPIO pin numbers that need to be excluded from the example. This may be necessary if some pins are already being used for other functions, e.g. if pins are being used for a UART connection during debugging.

Building and running the example

The `CMake` system, which is included with KasperskyOS Community Edition SDK, is used to build and run the example.

After starting the example, to generate interrupts and correctly finish the example, sequentially apply a short-duration 3.3 V voltage impulse to each of the GPIO pins (except for those specified in the `ExceptionPinArr` array).

See Building and running examples section.

gpio_echo example

This example demonstrates use of the GPIO driver to verify the input/output functionality of GPIO pins and the operation of interrupts for GPIO pins.

Example directory in the SDK

The code of the example and build scripts are available at the following path:

```
/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples/gpio_echo
```

List of programs

- The example includes system programs supplied in the KasperskyOS boot image and additional system programs and drivers required for working with the emulator or a specific hardware platform on which the solution will run. For more details, refer to Architecture and startup of a KasperskyOS-based solution.

  System programs are added to the solution image automatically during the build.

- `Client` – application that interacts with the GPIO driver. The program build script (the `./client/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

Security policy description

The `security.psl` file contains a description of a security policy for a KasperskyOS-based solution and is located in the `./einit/src` directory. The declarations in the PSL file are provided with comments that explain the purpose of these declarations.

Additional files

The `./resources/xdl/gpio_echo` directory contains the `Client.edl` file, which is the EDL description of the `Client` program.

The `./einit/CMakeLists.txt` file contains instructions for building a KasperskyOS boot image.

The `./einit/src/` directory contains the configuration files for the ServiceManager system program, which is responsible for installing and running KPA packages:

- The `initial_install.json` file contains a list of KPA packages that must be installed when KasperskyOS is started for the first time.
- The `system_services.json` file contains a list of installed programs that must be started each time KasperskyOS starts.

Operating scenario

The `Client` program performs the following actions:

- Initializes drivers: BSP and GPIO. Opens the GPIO port with the name specified in the `HW_MODULE_NAME` macro.
- Configures GPIO pins:
  - The pin with the number specified in the `GPIO_PIN_OUT` macro is configured as an output pin.
  - The pin with the number specified in the `GPIO_PIN_IN` macro is configured as an input pin with interrupts enabled during the state transition (`IN_MODE` macro).
- Sets the output pin to a logical one state and verifies that an interrupt event was registered from the input pin. Verifies that the state of the input pin corresponds to a logical one.
- Sets the output pin to a logical zero state and checks whether an interrupt event was received from the input pin. Verifies that the state of the input pin corresponds to a logical zero.
- Closes the GPIO port, deinitializes drivers, and terminates with the `EXIT_SUCCESS` code if there are no errors.

Building and running the example

The `CMake` system, which is included with KasperskyOS Community Edition SDK, is used to build and run the example.

For the `gpio_echo` example to work correctly, you must physically connect the output pin (the pin number is specified in the `GPIO_PIN_OUT` macro) to the input pin (`GPIO_PIN_IN`). This provides feedback so that a change in the state of an output pin triggers an interrupt event on the input pin.

See Building and running examples section.

kcfm_echo_server example

This example demonstrates integration of the KCFM component into a KasperskyOS-based web server where an application behavior signature is generated, packaged into a KPA package along with an ELF file, and compared with the actual behavior at startup to detect anomalies. The process includes deploying the package, starting the application through KCFM, and monitoring its operation to detect deviations from the specified signature.

Example directory in the SDK

The code of the example and build scripts are available at the following path:

```
/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples/kcfm_echo_server
```

List of programs

- The example includes system programs supplied in the KasperskyOS boot image and additional system programs and drivers required for working with the emulator or a specific hardware platform on which the solution will run. For more details, refer to Architecture and startup of a KasperskyOS-based solution.

  System programs are added to the solution image automatically during the build.

- `FlowChecker` – KCFM component module program that detects anomalies in the behavior of the `Server` application based on a security audit. If an anomaly is detected, `FlowChecker` reports it to `FlowMonitor`. The script for the example build ./kcfm/CMakeLists.txt contains instructions for packaging the program into a KPA package.

- `FlowMonitor` – KCFM component module program that provides control over the execution of the `Server` application by starting the application and processing the message regarding the detected anomaly. The script for the example build ./kcfm/CMakeLists.txt contains instructions for packaging the program into a KPA package.

- `WebServer` – monitored application that performs web server functions. The program build script (the `./web_server/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

- `Launcher` – application that starts KCFM application processes and applications for authentication and processing of commands. The program build script (the `./launcher/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

- `CommandProcessor` – application that processes commands from the web server, authenticates the user, and manages the session. The program build script (the `./command_processor/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

Generating a signature

The CMake command `kcfm_generate_signature` located in the ./web_server/CMakeLists.txt file determines which interfaces will be used when creating the CFG and signature of the application. The list of interfaces is specified in the `EDL` parameter. This example uses the interfaces `kl.core.Core`, `kl.VfsNet`, `kl.VfsEntity`, and `echo_server.CommandProcessor`, which are the interfaces for querying the OS kernel, the network subsystem, the file subsystem, and the CommandProcessor program, respectively.

Security policy description

The `security.psl` file contains a description of a security policy for a KasperskyOS-based solution and is located in the `./einit/src` directory. The declarations in the PSL file are provided with comments that explain the purpose of these declarations.

> The solution security policy description in this example allows any interaction between different processes and between processes and the kernel. This policy can be used only as a stub during the early stages of a solution development so that the security module does not interfere with interactions.

Additionally, the CMake command `kcfm_generate_default_policy()` has been added to the `web_server/CMakeLists.txt` file for building the `WebServer` program. This command automatically generates part of the solution security policy in the form of a PSL file based on the program source code. This PSL file lists and allows all IPC calls of core endpoints and VFS system program interfaces used in the program code. The security policy is formed so as to allow the program to access the IPC calls in use and deny access to all of the unused ones. The PSL file is created in the `./build/server/Server` directory and is included in the solution security policy description.

The CMake command `kcfm_generate_audit_policy()` has also been added to the `web_server/CMakeLists.txt` file to build the `WebServer` program. This command creates the security audit profiles necessary for the KCFM component to gain access to the security audit data of the monitored program. These audit profiles are added to the solution security policy.

Additional files

The `./resources/xdl/echo_server` directory contains the `WebServer.edl` file: EDL descriptions of the `WebServer` and `CommandProcessor` programs and their IDL interfaces.

The `./resources/hdd/etc` directory contains the `hosts` and `dhcpcd.conf` configuration files for the `VfsNet` and `Dhcpcd` programs, respectively, and the `resolv.conf` file, which contains data on the DNS server, and the `logrr_server_config.json` file, which contains the log level.

The `./resources/hdd/var/db/dhcpcd` directory is used by the `Dhcpcd` program to store network connection data.

The `./resources/hdd/www` directory contains the files used by the `WebServer` program while the web server is running.

The `./logrr/CMakeLists.txt` file contains instructions for building the logging subsystem.

The `./einit/CMakeLists.txt` file contains instructions for building a KasperskyOS boot image.

The `./einit/src/` directory contains the configuration files for the ServiceManager system program, which is responsible for installing and running KPA packages:

- The `initial_install.json` file contains a list of KPA packages that must be installed when KasperskyOS is started for the first time.
- The `initial_install_kcfm.json` file contains a list of KPA packages that must be installed when KasperskyOS is started for the first time when KCFM technology is enabled.
- The `system_services.json` file contains a list of installed programs that must be started each time KasperskyOS starts.

The `./einit/src/product_init_qemu.yaml` and `./einit/src/product_init_rpi4.yaml` files contain the startup parameters for the `DNetSrv, VfsNet, and UART` system programs and the logging subsystem in Init description format.

Operating scenario

1. The following CMake commands have been added to the `./files/echo_server/CMakeLists.txt` file to build the `WebServer` program:
   - `kcfm_generate_audit_policy()` to automatically generate part of the solution security policy.
   - `kcfm_generate_signature()` to build a control flow graph.
2. When building the `WebServer` program, a part of the information policy is generated in the file `build/_headers_/hello/Hello/kcfmDefaultPolicy.psl` and the files containing the control flow graph are generated in the `build/hello/kcfm_generated` directory.

   > Building the control flow graph in the kcfm_echo_server example may take a long time (up to 2 hours).

3. After building the solution image, the WebServer program starts a web server that is accessible at `http://localhost:1106` when running an example in QEMU. When started on RaspberryPI, the IP address is determined by the network connection settings of the device
4. When you go to the web server address, the `auth.html` web page opens in your browser for user authentication.

   > The correct authentication credentials are in the file named kcfm_echo_server/command_processor/src/session_manager.cpp

   When you enter the correct authentication information and click the `Login` button, the web server opens the local web page `command.html`. This page shows the token issued during authentication and the command input field.

   The entered command is processed by the `CommandProcessor` program. The result is displayed on the page. For the echo_server example, the result is the text of the entered command.

   To run a web_server program anomalous behavior check, which is built into the server source code and is based on the execution of parasitic activity triggered by a buffer overflow vulnerability, enter the "`hack@anomaly`" command. After entering this command, web_server performs parasitic activity (saves the path of the executable program to the hack.txt file).

The `FlowChecker` program detects the anomaly and notifies the `FlowMonitor` program, which restarts web_server.

Building and running the example

The CMake system, which is included with KasperskyOS Community Edition SDK, is used to build and run the example.

See Building and running examples section.

logrr example

This example demonstrates how to use the LogRR component to log the operation of programs in a KasperskyOS-based solution.

Example directory in the SDK

The code of the example and build scripts are available at the following path:

```
/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples/logrr
```

List of programs

- The example includes system programs supplied in the KasperskyOS boot image and additional system programs and drivers required for working with the emulator or a specific hardware platform on which the solution will run. For more details, refer to Architecture and startup of a KasperskyOS-based solution.

  System programs are added to the solution image automatically during the build.

- `Client` – application that sends messages to the log server. The `Client` program sends messages to the logging server by using the `logrr_cpp` library, which filters messages by their log level. The program build script (the `./client/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

- `Launcher` – application that waits for the network interfaces to initialize and then starts the `Client` program. The program build script (the `./launcher/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

Security policy description

The `security.psl` file contains a description of a security policy for a KasperskyOS-based solution and is located in the `./einit/src` directory. The declarations in the PSL file are provided with comments that explain the purpose of these declarations.

Additional files

The `./resources/xdl/logrr` directory contains the `Client.edl` file, which is the EDL description of the `Client` program.

The `./resources/hdd/etc` directory contains the configuration files for the `VfsNet`, `Dhcpcd`, and `Ntpd` programs: `hosts`, `dhcpcd.conf`, and `ntp.conf`, respectively.

The `./resources/hdd/var/db/dhcpcd` directory is used by the `Dhcpcd` program to store network connection data.

The `./einit/CMakeLists.txt` file contains instructions for building a KasperskyOS boot image.

The `./logrr/CMakeLists.txt` file contains instructions for building and running the logging subsystem. Two output channels are created: one to save the messages received from the logging server to a file and one to print them to the serial port (UART).

The `./einit/src/` directory contains the configuration files for the ServiceManager system program, which is responsible for installing and running KPA packages:

- The `initial_install.json` file contains a list of KPA packages that must be installed when KasperskyOS is started for the first time.
- The `system_services.json` file contains a list of installed programs that must be started each time KasperskyOS starts.

The `./einit/src/product_init_qemu.yaml`, `./einit/src/product_init_rpi4.yaml`, and `./einit/src/product_init_opirv2.yaml` files contain the startup parameters for the `DNetSrv, VfsNet, and UART` system programs and the logging subsystem in Init description format.

Operating scenario

The `Client` program performs the following actions:

1. Checks the availability of the network. If the network is not available, the program displays an error message and continues working without a network.
2. Waits to synchronize with NTP servers for 5 seconds.

3. Initializes the client side of the LogRR component.

4. Sends messages to the log using the LOG macro.

Building and running the example

The CMake system, which is included with KasperskyOS Community Edition SDK, is used to build and run the example.

See Building and running examples section.

pcre example

This example demonstrates use of the `pcre` library in KasperskyOS.

Example directory in the SDK

The code of the example and build scripts are available at the following path:

```
/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples/pcre
```

List of programs

- The example includes system programs supplied in the KasperskyOS boot image and additional system programs and drivers required for working with the emulator or a specific hardware platform on which the solution will run. For more details, refer to Architecture and startup of a KasperskyOS-based solution.

  System programs are added to the solution image automatically during the build.

- `Client` – application that uses the `pcre` library and prints the results to the console. The program build script (the `./client/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

Security policy description

The `security.psl` file contains a description of a security policy for a KasperskyOS-based solution and is located in the `./einit/src` directory. The declarations in the PSL file are provided with comments that explain the purpose of these declarations.

Additional files

The `./resources/xdl/pcre` directory contains the `Client.edl` file, which is the EDL description of the `Client` program.

The `./einit/CMakeLists.txt` file contains instructions for building a KasperskyOS boot image.

The `./einit/src/` directory contains the configuration files for the ServiceManager system program, which is responsible for installing and running KPA packages:

- The `initial_install.json` file contains a list of KPA packages that must be installed when KasperskyOS is started for the first time.
- The `system_services.json` file contains a list of installed programs that must be started each time KasperskyOS starts.

Operating scenario

The `Client` program performs the following actions:

- Creates a regular expression with a `Kaspersky` template.
- Prepares text for analysis.
- Searches for all occurrences of the word `Kaspersky` in the prepared text using the `FindAndConsume()` method.
- Prints the found matches with their sequence numbers to standard error output.
- Ends execution with the `EXIT_SUCCESS` code.

Building and running the example

The `CMake` system, which is included with KasperskyOS Community Edition SDK, is used to build and run the example.

See Building and running examples section.

I2c_ds1307_rtc example

This example demonstrates how to work with I2C (Inter-Integrated Circuit) to control a real-time clock module based on the DS1307Z chip in KasperskyOS.

Example directory in the SDK

The code of the example and build scripts are available at the following path:

```
/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples/i2c_ds1307_rtc
```

List of programs

- The example includes system programs supplied in the KasperskyOS boot image and additional system programs and drivers required for working with the emulator or a specific hardware platform on which the solution will run. For more details, refer to Architecture and startup of a KasperskyOS-based solution.

  System programs are added to the solution image automatically during the build.

- `I2cClient` – application that uses the I2C driver interface. – application that changes the current system time. The program build script (the `./i2c_client/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

Security policy description

The `security.psl` file contains a description of a security policy for a KasperskyOS-based solution and is located in the `./einit/src` directory. The declarations in the PSL file are provided with comments that explain the purpose of these declarations.

Additional files

The `./resources/xdl/i2c_ds1307_rtc` directory contains the `I2cClient.edl` file, which is the EDL description for the `I2cClient` program.

The `./einit/CMakeLists.txt` file contains instructions for building a KasperskyOS boot image.

The `./einit/src/` directory contains the configuration files for the ServiceManager system program, which is responsible for installing and running KPA packages:

- The `initial_install.json` file contains a list of KPA packages that must be installed when KasperskyOS is started for the first time.

- The `system_services.json` file contains a list of installed programs that must be started each time KasperskyOS starts.

The `./einit/src/product_init_qemu.yaml`, `./einit/src/product_init_rpi4.yaml`, and `./einit/src/product_init_opirv2.yaml` files contain the startup parameters for the `DNetSrv, VfsNet, and UART` system programs and the logging subsystem in Init description format.

Operating scenario

The `I2cClient` program performs the following actions:

1. Initializes the BSP driver and applies the predefined configuration of the I2C module, which configures output multiplexing for the specified I2C channel.
2. Initializes the I2C driver, lists the available I2C channels, and opens the I2C channel for operation.
3. Generates a buffer with fixed test values for the date and time, and writes the prepared buffer to the real-time clock registers.
4. Reads the current values of the real-time clock registers and compares the read data with the expected test values.
5. Closes the I2C channel and terminates with the `EXIT_SUCCESS` code if the data matches, otherwise it shows the `EXIT_FAILURE` code.

Building and running the example

The CMake system, which is included with KasperskyOS Community Edition SDK, is used to build and run the example.

For the example to work correctly, you must connect a DS1307Z chip-based real-time clock module to the I2C port.

Diagram for connecting the DS1307Z to Raspberry Pi:

- Connect SDA (Serial Data) to GPIO 0 (pin 27).
- Connect SCL (Serial Clock) to GPIO 1 (pin 28).
- Connect VCC to 5 V power (pin 2 or 4).
- Connect GND to Ground (pin 6).

Diagram for connecting the DS1307Z to OrangePi RV2:

- Connect SDA (Serial Data) to GPIO 51 (pin 3).
- Connect SCL (Serial Clock) to GPIO 52 (pin 5).
- Connect VCC to 5 V power (pin 2 or 4).
- Connect GND to Ground (pin 6).

See Building and running examples section.

iperf_separate_vfs example

This example demonstrates use of the `iperf` library in KasperskyOS.

Example directory in the SDK

The code of the example and build scripts are available at the following path:

```
/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples/iperf_separate_vfs
```

List of programs

- The example includes system programs supplied in the KasperskyOS boot image and additional system programs and drivers required for working with the emulator or a specific hardware platform on which the solution will run. For more details, refer to Architecture and startup of a KasperskyOS-based solution.

  System programs are added to the solution image automatically during the build.

- `Server` – application that performs the functions of a network server using the `iperf` library. The application initializes the network connection and runs the network performance test in server mode. The program build script (the `./server/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

Security policy description

The `security.psl` file contains a description of a security policy for a KasperskyOS-based solution and is located in the `./einit/src` directory. The declarations in the PSL file are provided with comments that explain the purpose of these declarations.

Additional files

The `./resources/xdl/iperf_separate_vfs` directory contains the `Server.edl` file, which is the EDL description of the `Server` program.

The `./einit/CMakeLists.txt` file contains instructions for building a KasperskyOS boot image.

The `./einit/src/` directory contains the configuration files for the ServiceManager system program, which is responsible for installing and running KPA packages:

- The `initial_install.json` file contains a list of KPA packages that must be installed when KasperskyOS is started for the first time.
- The `system_services.json` file contains a list of installed programs that must be started each time KasperskyOS starts.

The `./einit/src/product_init.yaml` file contains the startup parameters for the `VfsNet` and `DNetSrv` system programs in Init description format.

Operating scenario

The `Server` program performs the following actions:

- Waits for the availability of the network interface with the name specified by the `DEFAULT_INTERFACE` macro and with the timeout specified by the `DEFAULT_TIMEOUT` macro.
- Configures a network interface with the IP address specified by the `DEFAULT_ADDR` macro and the subnet mask specified by the `DEFAULT_MASK` macro.
- Prints a list of available network interfaces to standard error output.
- Creates a new iperf test using the `iperf_new_test()` function.
- Initializes test parameters with their default values using the `iperf_defaults()` function.
- Processes command line arguments to further customize test parameters using the `iperf_parse_arguments()` function.
- Runs the iperf test in server mode using the `run_test()` function.
- Writes a message about the start of testing to standard error output and displays a message about its completion after the end of the test.
- Frees resources of the iperf test using the `iperf_free_test()` function and exits the program with the `EXIT_SUCCESS` code upon successful termination.

Building and running the example

The `CMake` system, which is included with KasperskyOS Community Edition SDK, is used to build and run the example.

By default, the example uses network software emulation (SLIRP) in QEMU. If you configured TAP interfaces for QEMU, you need to change the network settings for starting QEMU (`QEMU_FLAGS` variable) in the `./einit/CMakeLists.txt` file to make sure that the example works correctly (for more details, see the comments in the file).

The example does not use DHCP, therefore the IP address of the network interface must be manually indicated in the code of the `Server` program (`./server/src/main.cpp`). SLIRP uses the default values.

The `iperf` library in the example is used in server mode. To connect to this server, install the `iperf3` program on the host machine and run it by using the `iperf3 -c localhost` command. If you configured TAP interfaces, indicate the current IP address instead of `localhost`.

The first startup of the example may take a long time because the `iperf` client uses `/dev/urandom` to fill packets with random data. To avoid this, run the `iperf` client with the `--repeating-payload` parameter.

See Building and running examples section.

uart example

This example demonstrates use of the Universal Asynchronous Receiver-Transmitter (UART) driver in KasperskyOS.

Example directory in the SDK

The code of the example and build scripts are available at the following path:

```
/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples/uart
```

List of programs

- The example includes system programs supplied in the KasperskyOS boot image and additional system programs and drivers required for working with the emulator or a specific hardware platform on which the solution will run. For more details, refer to Architecture and startup of a KasperskyOS-based solution.

  System programs are added to the solution image automatically during the build.

- `Client` – application that uses the UART driver interface. A full description of the UART driver interface is contained in the file `/opt/KasperskyOS-Community-Edition-<platform>-<version>/sysroot-*-kos/include/uart/uart.h`. The program build script (the `./client/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

Security policy description

The `security.psl` file contains a description of a security policy for a KasperskyOS-based solution and is located in the `./einit/src` directory. The declarations in the PSL file are provided with comments that explain the purpose of these declarations.

Additional files

The directory `./resources/xdl/uart` contains the EDL description for the `Client` program.

The `./einit/CMakeLists.txt` file contains instructions for building a KasperskyOS boot image.

The `./einit/src/` directory contains the configuration files for the ServiceManager system program, which is responsible for installing and running KPA packages:

- The `initial_install.json` file contains a list of KPA packages that must be installed when KasperskyOS is started for the first time.
- The `system_services.json` file contains a list of installed programs that must be started each time KasperskyOS starts.

The `./einit/src/product_init_qemu.yaml`, `./einit/src/product_init_rpi4.yaml`, and `./einit/src/product_init_opirv2.yaml` files contain the startup parameters for the `DNetSrv, VfsNet, and UART` system programs and the logging subsystem in Init description format.

Operating scenario

The `Client` program performs the following actions:

1. Moves logging to the user space to gain exclusive access to the UART port.
2. Initializes the UART driver.
3. Opens the port with the number specified by the `PORT_NAME` macro and configures it with the defined parameters.
4. Sends the string `Hello world with UART!` to the open port.
5. Closes the port, deinitializes the driver, and terminates the program with the `EXIT_SUCCESS` code if there are no errors. Otherwise it shows the `EXIT_FAILURE` code.

Building and running the example

The CMake system, which is included with KasperskyOS Community Edition SDK, is used to build and run the example.

See Building and running examples section.

spi_check_regs example

This example demonstrates how to work with the SPI (Serial Peripheral Interface) on the Sense HAT add-on board in KasperskyOS.

Example directory in the SDK

The code of the example and build scripts are available at the following path:

```
/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples/spi_check_regs
```

List of programs

- The example includes system programs supplied in the KasperskyOS boot image and additional system programs and drivers required for working with the emulator or a specific hardware platform on which the solution will run. For more details, refer to Architecture and startup of a KasperskyOS-based solution.

  System programs are added to the solution image automatically during the build.

- `Client` – application that uses the SPI driver interface. The program build script (the `./client/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

Security policy description

The `security.psl` file contains a description of a security policy for a KasperskyOS-based solution and is located in the `./einit/src` directory. The declarations in the PSL file are provided with comments that explain the purpose of these declarations.

Additional files

The directory `./resources/xdl/spi_check_regs` contains the `Client.edl` file, which is the EDL description for the `Client` program.

The `./einit/CMakeLists.txt` file contains instructions for building a KasperskyOS boot image.

The `./einit/src/` directory contains the configuration files for the ServiceManager system program, which is responsible for installing and running KPA packages:

- The `initial_install.json` file contains a list of KPA packages that must be installed when KasperskyOS is started for the first time.
- The `system_services.json` file contains a list of installed programs that must be started each time KasperskyOS starts.

The `./einit/src/product_init_rpi4.yaml` and `./einit/src/product_init_opirv2.yaml` files contain the startup parameters for SPI system programs in Init description format.

Operating scenario

The `Client` program performs the following actions:

1. Initializes the following drivers: PinCtrl for OrangePi RV2 or BSP for Raspberry Pi 4 B, and GPIO and SPI drivers. Opens the GPIO port.
2. Configures the GPIO pin specified by the `HW_GPIO_RESET_PIN_NUM` macro for output and sets it to the logical 1 state (deactivation of the RESET signal of the ATTiny microcontroller).
3. Opens the SPI channel defined by the `HW_SPI_MODULE_CHANNEL` macro, receives the specifications of this channel, and prints them to standard error. Then it sets the operating mode for this channel as defined in the `cfg` structure.
4. Sets the GPIO pin defined by the `HW_GPIO_RESET_PIN_NUM` macro to the logical zero state (activation of the RESET signal of the ATTiny microcontroller).
5. Sends the `Programming Enable` command to the ATTiny microcontroller via the SPI channel. The signature bytes of the microcontroller are additionally read for OrangePi RV2.
6. Returns the GPIO pin defined by the `HW_GPIO_RESET_PIN_NUM` macro to the logical 1 state.
7. Prints the SPI exchange results to standard error and compares them with the expected values.
8. Closes the SPI channel and GPIO port.

Building and running the example

The CMake system, which is included with KasperskyOS Community Edition SDK, is used to build and run the example.

For the example to work correctly, do the following:

- Connect the Sense HAT add-on board to the 40-pin GPIO connector of the selected hardware platform.

  Full alignment of all pins is required on Raspberry Pi 4 B. On OrangePi RV2, manual connection of the pins is required according to the connection diagram in the table below.

Diagram for connecting Sense HAT to hardware platforms

Pin	Sense HAT	Raspberry Pi 4 B	OrangePi RV2
17	3V3 Power	3V3 power	3v3 Power
19	GPIO10 (SPI0 MOSI)	GPIO 10 (MOSI)	GPIO77
20	Ground	Ground	Ground
21	GPIO9 (SPI0 MISO)	GPIO 9 (MISO)	GPIO78
22	GPIO25 (Atmel Prog Reset)	GPIO 25	GPIO49
23	GPIO11 (SPI0 SCLK)	GPIO 11 (SCLK)	GPIO75
24	GPIO8 (Atmel Chip Select)	GPIO 8 (CE0)	GPIO76

See Building and running examples section.

barcode_scanner example

This example demonstrates use of a USB (Universal Serial Bus) driver in KasperskyOS using the `libevdev` library.

Example directory in the SDK

The code of the example and build scripts are available at the following path:

```
/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples/barcode_scanner
```

List of programs

- The example includes system programs supplied in the KasperskyOS boot image and additional system programs and drivers required for working with the emulator or a specific hardware platform on which the solution will run. For more details, refer to Architecture and startup of a KasperskyOS-based solution.

  System programs are added to the solution image automatically during the build.

- `BarcodeScanner` – application that interacts with a barcode scanner. The program build script (the `./barcode_scanner/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

Security policy description

The `security.psl` file contains a description of a security policy for a KasperskyOS-based solution and is located in the `./einit/src` directory. The declarations in the PSL file are provided with comments that explain the purpose of these declarations.

Additional files

The `./einit/CMakeLists.txt` file contains instructions for building a KasperskyOS boot image.

The `./einit/src/` directory contains the configuration files for the ServiceManager system program, which is responsible for installing and running KPA packages:

- The `initial_install.json` file contains a list of KPA packages that must be installed when KasperskyOS is started for the first time.
- The `system_services.json` file contains a list of installed programs that must be started each time KasperskyOS starts.

The `./einit/src/product_init_rpi4.yaml` and `./einit/src/product_init_opirv2.yaml` files contain the startup parameters for drivers in Init description format.

Operating scenario

In this example, the `BarcodeScanner` program uses the `libevdev` library to log keyboard events because most barcode scanners mimic the behavior of keyboard devices. The program waits for signals from the barcode scanner and prints the obtained data to `stderr`.

Building and running the example

The `CMake` system, which is included with KasperskyOS Community Edition SDK, is used to build and run the example.

For the example to work correctly, you must connect a barcode scanner running in keyboard emulation mode (such as Zebra Symbol LS2208) to the USB port.

See Building and running examples section.

watchdog_system_reset example

This example demonstrates use of the Watchdog driver in a KasperskyOS-based solution.

Example directory in the SDK

The code of the example and build scripts are available at the following path:

```
/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples/watchdog_system_reset
```

List of programs

- The example includes system programs supplied in the KasperskyOS boot image and additional system programs and drivers required for working with the emulator or a specific hardware platform on which the solution will run. For more details, refer to Architecture and startup of a KasperskyOS-based solution.

  System programs are added to the solution image automatically during the build.

- `Client` – application that uses the `Watchdog` driver interface to interact with the watchdog timer. The program build script (the `./client/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package. The client library of the `Watchdog` driver is statically linked to the `Client` program.

Security policy description

The `security.psl` file contains a description of a security policy for a KasperskyOS-based solution and is located in the `./einit/src` directory. The declarations in the PSL file are provided with comments that explain the purpose of these declarations.

Additional files

The `./resources/xdl/watchdog_system_reset` directory contains the `Client.edl` file, which is the EDL description for the `Client` program.

The `./einit/CMakeLists.txt` file contains instructions for building a KasperskyOS boot image.

The `./einit/src/` directory contains the configuration files for the ServiceManager system program, which is responsible for installing and running KPA packages:

- The `initial_install.json` file contains a list of KPA packages that must be installed when KasperskyOS is started for the first time.

- The `system_services.json` file contains a list of installed programs that must be started each time KasperskyOS starts.

The `./einit/src/product_init_rpi4.yaml` file contains the startup parameters for the `VfsNet` and `DNetSrv` system programs in Init description format.

Operating scenario

The `Client` program performs the following actions:

1. Gets the current `Watchdog` driver settings and prints them to standard error.

2. Changes the default value of the timer to a new value and starts the timer.

3. Resets the timer multiple times.

4. Waits for the system to restart when the timer is triggered.

Building and running the example

The CMake system, which is included with KasperskyOS Community Edition SDK, is used to build and run the example.

See Building and running examples section.

shared_libs example

This example demonstrates use of static and dynamic libraries in KasperskyOS.

Example directory in the SDK

The code of the example and build scripts are available at the following path:

```
/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples/shared_libs
```

List of programs

- The example includes system programs supplied in the KasperskyOS boot image and additional system programs and drivers required for working with the emulator or a specific hardware platform on which the solution will run. For more details, refer to Architecture and startup of a KasperskyOS-based solution.

  System programs are added to the solution image automatically during the build.

- `Client` – application that uses static and dynamic libraries. The program build script (the `./client/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

Security policy description

The `security.psl` file contains a description of a security policy for a KasperskyOS-based solution and is located in the `./einit/src` directory. The declarations in the PSL file are provided with comments that explain the purpose of these declarations.

Additional files

The `./resources/xdl/shared_libs` directory contains the `Client.edl` file, which is the EDL description of the `Client` program.

The `./einit/CMakeLists.txt` file contains instructions for building a KasperskyOS boot image.

The `./einit/src/` directory contains the configuration files for the ServiceManager system program, which is responsible for installing and running KPA packages:

- The `initial_install.json` file contains a list of KPA packages that must be installed when KasperskyOS is started for the first time.
- The `system_services.json` file contains a list of installed programs that must be started each time KasperskyOS starts.

Operating scenario

In the example, the `Client` program performs the following actions:

- Calls a function from the `hello_s` static library.
- Calls a function from the `hello_d1` dynamic library that is linked together with the program and loaded into memory when the process is started.
- Calls a function from the `hello_d2` dynamic library that is loaded into memory when calling the `dlopen()` function of the POSIX interface.

Building and running the example

The `CMake` system, which is included with KasperskyOS Community Edition SDK, is used to build and run the example.

See Building and running examples section.

pal_tests example

This example demonstrates use of the Policy Assertion Language (PAL) when writing tests for a KasperskyOS-based solution security policy. These tests in PAL let you check the solution security policy even before starting development of the software code based on the formal specifications of solution components.

Example directory in the SDK

The code of the example and build scripts are available at the following path:

```
/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples/pal_tests
```

List of programs

The example does not contain the programs themselves, but only their formal descriptions. The tests check the security events of the following programs:

- `WebServer` – application that provides the user with access to the `Service` program via the web interface.
- `Service` – application that handles user requests and returns the results of request completion.
- `UserManager` – application that implements user account management and user authorization.

Security policy description

The security policy description files for a KasperskyOS-based solution are located at the path `./einit/src`.

The `security.psl` file contains description of the solution security policy. Through the `use declaration`, this top-level file includes the following parts of a solution security policy description provided by PSL files: `core.psl`, `user_mngr.psl`, `service.psl`, `vfs_net.psl`, and `vfs_entity.psl`.

The `*_tests.psl` files contain sets of tests.

Additional files

The `./resources/edl` and `./resources/idl` directories contain the EDL and IDL descriptions for the `WebServer`, `Service` and `UserManager` programs.

Operating scenario

The sets of tests described in the `*_tests.psl` files are executed to check whether each program from the list complies with the defined security policies. The results of tests are displayed in standard output.

Building and running the example

See "Creating and performing tests for a KasperskyOS-based solution security policy".

hello_from_rust example

This example demonstrates how to let your KasperskyOS-based solution include a simple program that was developed based on Rust and built using the Cargo build system and package manager.

Example directory in the SDK

The code of the example and build scripts are available at the following path:

```
/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples/hello_from_rust
```

List of programs

- The example includes system programs supplied in the KasperskyOS boot image and additional system programs and drivers required for working with the emulator or a specific hardware platform on which the solution will run. For more details, refer to Architecture and startup of a KasperskyOS-based solution.

  System programs are added to the solution image automatically during the build.

- `Hello` – application that writes a message to standard error output. This program was developed using the Rust language. The program build script (the `./hello/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

Security policy description

The `security.psl` file contains a description of a security policy for a KasperskyOS-based solution and is located in the `./einit/src` directory. The declarations in the PSL file are provided with comments that explain the purpose of these declarations.

The solution security policy description in this example allows any interaction between different processes and between processes and the kernel. This solution security policy is used to simplify the example. It would be unacceptable to apply such a policy in a real-world solution.

Additional files

The directory `./resources` contains an EDL description of the `Hello` program.

The `./einit/CMakeLists.txt` file contains instructions for building a KasperskyOS boot image.

The `./einit/src/` directory contains the configuration files for the ServiceManager system program, which is responsible for installing and running KPA packages:

- The `initial_install.json` file contains a list of KPA packages that must be installed when KasperskyOS is started for the first time.
- The `system_services.json` file contains a list of installed programs that must be started each time KasperskyOS starts.

Operating scenario

The `Hello` program prints the message `Hello, world!` to standard error.

Building and running the example

The CMake build system from the KasperskyOS SDK is used during the solution build. The `Hello` program is built using the Cargo build system, which is also included in the SDK. During the solution build process, the `add_custom_command()` command in the `./hello/CMakeLists.txt` file executes the command `cargo build`, which builds the executable file of the `Hello` program. This file is included in a solution as an imported executable file via the `add_executable()` command.

See Building and running examples section.

hello_corrosion example

This example demonstrates how to let your solution include a simple program that was developed based on Rust and uses Corrosion, which is a set of libraries for integrating Rust into CMake projects.

Example directory in the SDK

The code of the example and build scripts are available at the following path:

```
/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples/hello_corrosion
```

List of programs

- The example includes system programs supplied in the KasperskyOS boot image and additional system programs and drivers required for working with the emulator or a specific hardware platform on which the solution will run. For more details, refer to Architecture and startup of a KasperskyOS-based solution.

  System programs are added to the solution image automatically during the build.

- `Hello` – application that writes a message to standard error output. This program was developed using the Rust language. The program build script (the `./hello/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

Security policy description

The `security.psl` file contains a description of a security policy for a KasperskyOS-based solution and is located in the `./einit/src` directory. The declarations in the PSL file are provided with comments that explain the purpose of these declarations.

The solution security policy description in this example allows any interaction between different processes and between processes and the kernel. This solution security policy is used to simplify the example. It would be unacceptable to apply such a policy in a real-world solution.

Additional files

The directory `./resources` contains an EDL description of the `Hello` program.

The `./einit/CMakeLists.txt` file contains instructions for building a KasperskyOS boot image.

The `./einit/src/` directory contains the configuration files for the ServiceManager system program, which is responsible for installing and running KPA packages:

- The `initial_install.json` file contains a list of KPA packages that must be installed when KasperskyOS is started for the first time.

- The `system_services.json` file contains a list of installed programs that must be started each time KasperskyOS starts.

Operating scenario

The `Hello` program prints the message `Hello, world!` to standard error.

Building and running the example

The CMake build system from the KasperskyOS SDK is used during the solution build.

To synchronize the target platform for building Rust-based source code with the target platform of the C compiler, you must add the following command to the `./CMakeList.txt` file:

```
set (Rust_CARGO_TARGET ${CMAKE_C_COMPILER_TARGET})
```

To use Corrosion in your solution, you must add the following command to this same file:

```
find_package (Corrosion REQUIRED)
```

To integrate Rust-based source code into your solution, you must call the following function in the `./hello/CMakeList.txt` file:

```
corrosion_import_crate (MANIFEST_PATH ${CMAKE_SOURCE_DIR}/hello/Cargo.toml)
```

This function compiles the `hello` executable file that will be included in the solution.

See Building and running examples section.

mass_storage example

This example demonstrates use of the UsbMassStorage driver in KasperskyOS-based solutions to work with an external USB drive connected to the hardware platform's USB port.

Example directory in the SDK

The code of the example and build scripts are available at the following path:

```
/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples/mass_storage
```

List of programs

- The example includes system programs supplied in the KasperskyOS boot image and additional system programs and drivers required for working with the emulator or a specific hardware platform on which the solution will run. For more details, refer to Architecture and startup of a KasperskyOS-based solution.

  System programs are added to the solution image automatically during the build.

- `Client` is an application that detects the connection of an external USB storage device, mounts its primary partition with the `ext4` file system in the `/mnt` directory, and checks the performance of the file system by performing test operations (creating, writing, reading, and deleting a file). The program build script (the `./client/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

- `FileVfs` implements the server portion of the virtual file system, thereby providing the interface used for interaction with file systems and block devices.

Security policy description

The `security.psl` file contains a description of a security policy for a KasperskyOS-based solution and is located in the `./einit/src` directory. The declarations in the PSL file are provided with comments that explain the purpose of these declarations.

Additional files

The directory `./resources/xdl/mass_storage` contains EDL descriptions for the `Client` and `FileVfs` programs.

The `./einit/CMakeLists.txt` file contains instructions for building a KasperskyOS boot image.

The `./einit/src/` directory contains the configuration files for the ServiceManager system program, which is responsible for installing and running KPA packages:

- The `initial_install.json` file contains a list of KPA packages that must be installed when KasperskyOS is started for the first time.

- The `system_services.json` file contains a list of installed programs that must be started each time KasperskyOS starts.

The `./einit/src/product_init_rpi4.yaml` and `./einit/src/product_init_opirv2.yaml` files contain the startup parameters for the `FileVfs` program in Init description format.

Operating scenario

The `Client` program interacts with the `FileVfs` program, which acts as a VFS server through an IPC interface. `FileVfs` mounts the `ext4` file system from an external USB drive to the `/mnt` directory. To test the mount functionality, the `Client` sequentially performs the following operations: creates a file named `new_file`, writes the `test_message` string to it, reads the contents, and deletes the file. The results of each operation are printed to the standard output stream.

Building and running the example

To prepare the USB drive for use in this example, run the following commands:

```
# In the following commands /dev/sde is the block device name of the connected USB drive
parted /dev/sde mklabel msdos
parted /dev/sde mkpart primary ext4 0% 100%
mkfs.ext4 -L flash /dev/sde1
```

See Building and running examples section.

framebuffer_qr_writer example

The example demonstrates how to set up a graphical user interface (GUI) in a KasperskyOS-based solution using a set of cross-platform libraries: SDL2 for managing the application window and processing user data input, Nuklear for building the graphical interface, Skia as a graphics rendering engine, and ZXing for generating QR codes.

Example directory in the SDK

The code of the example and build scripts are available at the following path:

```
/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples/framebuffer_qr_writer
```

List of programs

- The example includes system programs supplied in the KasperskyOS boot image and additional system programs and drivers required for working with the emulator or a specific hardware platform on which the solution will run. For more details, refer to Architecture and startup of a KasperskyOS-based solution.

  System programs are added to the solution image automatically during the build.

- `QrWriter` – application that converts user-entered text into a QR code and displays it in a graphical interface. The program build script (the `./qr_writer/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

Security policy description

The `security.psl` file contains a description of a security policy for a KasperskyOS-based solution and is located in the `./einit/src` directory. The declarations in the PSL file are provided with comments that explain the purpose of these declarations.

The solution security policy description in this example allows any interaction between different processes and between processes and the kernel. This policy can be used only as a stub during the early stages of a solution development so that the security module does not interfere with interactions. It would be unacceptable to apply such a policy in a real-world KasperskyOS-based solution.

Additional files

The `./resources/hdd/etc` directory includes the `ProggyClean.ttf` file, which contains information about the font (name, size, style). This file is used to display text in the graphical user interface.

The `./einit/CMakeLists.txt` file contains instructions for building a KasperskyOS boot image.

The `./einit/src/` directory contains the configuration files for the ServiceManager system program, which is responsible for installing and running KPA packages:

- The `initial_install.json` file contains a list of KPA packages that must be installed when KasperskyOS is started for the first time.
- The `system_services.json` file contains a list of installed programs that must be started each time KasperskyOS starts.

The `./einit/src/product_init_rpi4.yaml` file contains the startup parameters for drivers in Init description format.

Operating scenario

The `QrWriter` program performs the following actions:

1. Uses the SDL2 library to initialize the video subsystem and create a full-screen application window.
2. Uses the Skia library to configure the graphics context based on the window surface information received from SDL2.
3. Uses the Nuklear library to initialize the graphical interface via the nuklear_skia implementation.

4. In an infinite loop:

    - Handles input: uses SDL2 to poll the system event queue and uses Nuklear to convert events into graphical interface state changes.
    - Renders the graphical interface: uses Nuklear to generate commands for rendering controls in Immediate Mode.
    - Generates a QR code: when the button is clicked, it uses the ZXing library to convert the text into a monochrome bitmap image.
    - Renders a frame: uses the Skia library to execute drawing commands and generate the final image in the frame buffer.
    - Displays on screen: uses SDL2 to update the window surface with the contents of the frame buffer.

Building and running the example

The CMake system, which is included with KasperskyOS Community Edition SDK, is used to build and run the example.

See Building and running examples section.

dump_collector example

This example demonstrates the mechanism for creating a crash dump in a KasperskyOS-based solution.

Example directory in the SDK

The code of the example and build scripts are available at the following path:

```
/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples/dump_collector
```

List of programs

- The example includes system programs supplied in the KasperskyOS boot image and additional system programs and drivers required for working with the emulator or a specific hardware platform on which the solution will run. For more details, refer to Architecture and startup of a KasperskyOS-based solution.

  System programs are added to the solution image automatically during the build.

- `Application` – the application that crashes. The program build script (the `./application/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

Security policy description

The `security.psl` file contains a description of a security policy for a KasperskyOS-based solution and is located in the `./einit/src` directory. The declarations in the PSL file are provided with comments that explain the purpose of these declarations.

Additional files

The `./resources/xdl/application/Application.edl` file is an EDL description of the Application program.

The `./einit/CMakeLists.txt` file contains instructions for building a KasperskyOS boot image.

The `./einit/src/` directory contains the configuration files for the ServiceManager system program, which is responsible for installing and running KPA packages:

- The `initial_install.json` file contains a list of KPA packages that must be installed when KasperskyOS is started for the first time.
- The `system_services.json` file contains a list of installed programs that must be started each time KasperskyOS starts.

Operating scenario

```
The Application program is started at solution startup by the ServiceManager system program as it is listed in the
system_services.json file.
```

The started process performs actions that lead to its abnormal termination. When the process terminates abnormally, a crash dump is generated automatically. This dump contains diagnostic data such as the stack trace, the state of processor registers, information about threads, and the memory map of the process. The generated crash dump is printed to the console using the LogRR component.

Building and running the example

The CMake system, which is included with KasperskyOS Community Edition SDK, is used to build and run the example.

See Building and running examples section.

kpa_example

This example demonstrates how to work with KPA packages in a KasperskyOS-based solution:

- Downloading a KPA package from an HTTP server
- Building a KPA package from source components using functions of the kpa library
- Pre-installing a KPA package into a built image of a KasperskyOS-based solution using the kl_install_packages() CMake command
- Installing a KPA package into an operational KasperskyOS-based solution using the PackageManager component
- Getting data on a KPA package manifest
- Starting a process from an executable file installed from a KPA package and getting information about the process and stopping it using the ExecutionManager component

IPC connections between programs are established dynamically using the DCM system program, in which servers publish their interfaces and clients can connect to previously published interfaces. Please note that the same program can act as a client and server at the same time and interact with DCM as both a client and a server.

Example directory in the SDK

The code of the example and build scripts are available at the following path:

```
/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples/kpa_example
```

List of programs

- The example includes system programs supplied in the KasperskyOS boot image and additional system programs and drivers required for working with the emulator or a specific hardware platform on which the solution will run. For more details, refer to Architecture and startup of a KasperskyOS-based solution.

  System programs are added to the solution image automatically during the build.

- `Client` – application that works with three KPA packages: `DownloadedApp.kpa` (downloaded from an HTTP server), `Application.kpa` (built from source components and installed into an operational solution), `PreinstalledApp.kpa` (built from source components and pre-installed into a built solution image). The program build script (the `./client/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

Security policy description

The `security.psl` file contains a description of a security policy for a KasperskyOS-based solution and is located in the `./einit/src` directory. The declarations in the PSL file are provided with comments that explain the purpose of these declarations.

The solution security policy description in this example allows any interaction between different processes and between processes and the kernel. This policy can be used only as a stub during the early stages of a solution development so that the security module does not interfere with interactions. It would be unacceptable to apply such a policy in a real-world KasperskyOS-based solution.

Additional files

The `./resources/xdl/kpa_example` directory contains the `Application.edl`, `Client.edl`, `DownloadedApp.edl`, and `PreinstalledApp.edl` files, which are EDL descriptions of programs.

- The `./resources/hdd/etc` directory contains the configuration files for the `VfsNet` and `Dhcpcd` programs: `hosts` and `dhcpcd.conf`, respectively.
- The `text.txt` file is located in the `./resources/packages/application/files` directory. This file is a resource for the KPA package named `Application.kpa`.

The `./einit/CMakeLists.txt` file contains instructions for building a KasperskyOS boot image.

The `./einit/src/` directory contains the configuration files for the ServiceManager system program, which is responsible for installing and running KPA packages:

- The `initial_install.json` file contains a list of KPA packages that must be installed when KasperskyOS is started for the first time.
- The `system_services.json` file contains a list of installed programs that must be started each time KasperskyOS starts.

The `./einit/src/product_init.yaml` file contains the startup parameters for the `VfsNet` and `DNetSrv` system programs in Init description format.

Operating scenario

The `Client` program performs the following actions:

1. Checks the availability of the network. If the network is not available, it sends a message to the log using the LOG macro of the LogRR component and terminates.
2. Downloads the KPA package named DownloadedApp.kpa from an HTTP server.

3. Sends a message to the log containing a list of available KPA packages: `Application.kpa` and `DownloadedApp.kpa`. The KPA package named `Application.kpa` is built when a solution is built in the `./application/CMakeLists.txt` file.

4. Uses the `CreatePackageManager()` and `GetPackageController()` functions to get pointers to instances of the `IPackageManager` and `IPackageController` interfaces, respectively, which are required for working with the PackageManager component.

5. Installs the KPA package named `Application.kpa` using the `InstallPackage()` function of the `IPackageController` interface.

6. Sends a message to the log containing a list of unique identifiers of the installed KPA packages: `kpa_example.application` and `kpa_example.preinstalled_app`. The KPA package named `PreinstalledApp.kpa` is built and pre-installed into the built solution image in the `./preinstalled_app/CMakeLists.txt` file.

7. Sends a message to the log regarding the receipt of data about the `Application.kpa` package manifest by using the `GetManifest()` function of the `IPackageController` interface.

8. Gets the pointer to an instance of the IExecutionManager interface by using the `DcmConnect()` function. The DCM system program is used to connect to the ExecutionManager process.

9. Gets access to the IProcessControl interface using a pointer to an instance of the `IExecutionManager` interface.

10. Starts a process from an executable file supplied in the `Application.kpa` package. Receives the states of the running process in a loop using the `GetProcessState()` function of the `IProcessControl` interface and sends messages about the received states to the log. The process terminates on its own.

11. Starts a process from an executable file supplied in the `Application.kpa` package with a delay of 60 seconds.

12. Gets access to the ISystemController interface using a pointer to an instance of the `IExecutionManager` interface. Gets information about the processes started using the ExecutionManager component using the `GetSystemInfo()` function of the `ISystemController` interface and sends messages containing the received information to the log. After 5 seconds, forcibly terminates the running process by using the `StopProcess()` function of the `IProcessControl` interface.

Building and running the example

The CMake system, which is included with KasperskyOS Community Edition SDK, is used to build and run the example.

For the QEMU emulator, you must do the following after building the example but before running it:

1. Additionally build the KPA package named `DownloadedApp.kpa`. (The source code for the build is in the file named `./downloaded_kpa/CMakeLists.txt`.) To build the KPA package, run the following command:

```
cmake --build build --target DownloadedKpa
```

2. Start an HTTP server that will provide access to files from the `./build/kpa/` directory on port `8080`. To do so, in a separate terminal session, run the following command:

   ```
   python3 -m http.server -d ./build/kpa/ 8080
   ```

   To check the availability of the KPA package named `DownloadedApp.kpa`, open the page `http://localhost:8080` in your browser and make sure that the file is listed there.

For target hardware platforms, you must start the HTTP server before running the image containing the example.

See Building and running examples section.

uart_toybox example

This example demonstrates how to work with the Toybox component in a KasperskyOS-based solution.

Example directory in the SDK

The code of the example and build scripts are available at the following path:

```
/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples/uart_toybox
```

List of programs

- The example includes system programs supplied in the KasperskyOS boot image and additional system programs and drivers required for working with the emulator or a specific hardware platform on which the solution will run. For more details, refer to Architecture and startup of a KasperskyOS-based solution.

  System programs are added to the solution image automatically during the build.

- `Launcher` – application that waits for the network interfaces to initialize and then starts the `Toybox` program. The program build script (the `./launcher/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

- `Toybox` – application that implements the Toybox component. The program build script (the `./toybox/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

Security policy description

The `security.psl` file contains a description of a security policy for a KasperskyOS-based solution and is located in the `./einit/src` directory. The declarations in the PSL file are provided with comments that explain the purpose of these declarations.

The solution security policy description in this example allows any interaction between different processes and between processes and the kernel. This policy can be used only as a stub during the early stages of a solution development so that the security module does not interfere with interactions. It would be unacceptable to apply such a policy in a real-world KasperskyOS-based solution.

Additional files

The `./resources/xdl/exec_app/ExecApp.edl` file is the EDL description for the class of processes that are started when the `exec` command of the Toybox component is executed.

The `./resources/xdl/toybox` directory contains the `Launcher.edl` file, which is the EDL description of the `Launcher` program.

The `./resources/hdd/etc` directory contains the configuration files for the `Vfs` and `Dhcpcd` programs: `hosts` and `dhcpcd.conf`, respectively.

The `./resources/hdd/var/db/dhcpcd` directory is used by the `Dhcpcd` program to store network connection data.

The `./einit/CMakeLists.txt` file contains instructions for building a KasperskyOS boot image.

The `./einit/src/` directory contains the configuration files for the ServiceManager system program, which is responsible for installing and running KPA packages:

- The `initial_install.json` file contains a list of KPA packages that must be installed when KasperskyOS is started for the first time.
- The `system_services.json` file contains a list of installed programs that must be started each time KasperskyOS starts.

The `./einit/src/product_init_qemu.yaml`, `./einit/src/product_init_rpi4.yaml`, and `./einit/src/product_init_opirv2.yaml` files contain the startup parameters for the `DNetSrv, VfsNet, and UART` system programs and the logging subsystem in Init description format.

Operating scenario

The `Launcher` program starts the Toybox program installed from the KPA package when the solution is started.

After running the `Launcher` program, the terminal will go into interactive mode, as indicated by the display of the standard command line prompt `$`. In this mode, the commands described in the Toybox component become available.

Building and running the example

The CMake system, which is included with KasperskyOS Community Edition SDK, is used to build and run the example.

See Building and running examples section.

perfcnt example

This example demonstrates the use of performance counters in a KasperskyOS-based solution.

Example directory in the SDK

The code of the example and build scripts are available at the following path:

```
/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples/perfcnt
```

List of programs

- The example includes system programs supplied in the KasperskyOS boot image and additional system programs and drivers required for working with the emulator or a specific hardware platform on which the solution will run. For more details, refer to Architecture and startup of a KasperskyOS-based solution.

  System programs are added to the solution image automatically during the build.

- `Worker` – application that performs calculations in a loop, periodically loading the processor and using memory. The program build script (the `./worker/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

- `Monitor` – application that gets the performance counter values for the `Worker` program. The program build script (the `./monitor/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

- `Launcher` – application that starts the `Worker` and `Monitor` programs in the necessary `order`. The program build script (the `./launcher/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

Security policy description

The `./einit/src/security.psl` file contains description of the solution security policy. The declarations in the PSL file are provided with comments that explain the purpose of these declarations.

Additional files

The `./resources/xdl/perfcnt` directory contains EDL descriptions of the `Worker` and `Monitor` programs.

The `./einit/CMakeLists.txt` file contains instructions for building a KasperskyOS boot image.

The `./einit/src/` directory contains the configuration files for the ServiceManager system program, which is responsible for installing and running KPA packages:

- The `initial_install.json` file contains a list of KPA packages that must be installed when KasperskyOS is started for the first time.
- The `system_services.json` file contains a list of installed programs that must be started each time KasperskyOS starts.

Operating scenario

The `Monitor` program performs the following actions:

- Uses the stat_api.h API to get statistical information about the use of CPU time and memory, and about processes and threads.
- Uses the value_api.h API to get the values of performance counters.
- Prints the received data to standard error.

Building and running the example

The CMake system, which is included with KasperskyOS Community Edition SDK, is used to build and run the example.

See Building and running examples section.

vfs_relay example

This example demonstrates how to use the VFS relay mechanism to separate data streams among multiple VFS system programs.

The rules for selecting various VFS programs for performing file operations are implemented in the `RelayVfsServer` program code as follows:

- Operations with files in the `/local_vfs_dir` directory are performed locally by the `RelayVfsServer` program.
- Operations with files in the `/remote_vfs_dir` directory are redirected to the `VfsServer` program via IPC.

Example directory in the SDK

The code of the example and build scripts are available at the following path:

```
/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples/vfs_relay
```

List of programs

- The example includes system programs supplied in the KasperskyOS boot image and additional system programs and drivers required for working with the emulator or a specific hardware platform on which the solution will run. For more details, refer to Architecture and startup of a KasperskyOS-based solution.

  System programs are added to the solution image automatically during the build.
- `Client` – application that performs operations with files. The program build script (the `./client/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.
- `RelayVfsServer` – VFS server system program that implements the VFS relay mechanism.
- `VfsServer` – VFS server system program that is connected to the `RelayVfsServer` program as a backend.

Security policy description

The `security.psl` file contains a description of a security policy for a KasperskyOS-based solution and is located in the `./einit/src` directory. The declarations in the PSL file are provided with comments that explain the purpose of these declarations.

The solution security policy description in this example allows any interaction between different processes and between processes and the kernel. This policy can be used only as a stub during the early stages of a solution development so that the security module does not interfere with interactions. It would be unacceptable to apply such a policy in a real-world KasperskyOS-based solution.

Additional files

The directory `./resources/xdl/relay_vfs` contains EDL descriptions for the `Client`, `RelayVfsServer`, and `VfsServer`. programs.

The `./einit/CMakeLists.txt` file contains instructions for building a KasperskyOS boot image.

The `./einit/src/` directory contains the configuration files for the ServiceManager system program, which is responsible for installing and running KPA packages:

- The `initial_install.json` file contains a list of KPA packages that must be installed when KasperskyOS is started for the first time.
- The `system_services.json` file contains a list of installed programs that must be started each time KasperskyOS starts.

The `./einit/src/product_init.yaml` file contains the startup parameters for the `RelayVfsServer` and `VfsServer` system programs in Init description format.

Operating scenario

The `Client` program performs the following actions:

1. Creates two directories: `/local_vfs_dir` and `/remote_vfs_dir`.
2. Conducts tests that create, read, and write files in each of the created directories.

   In this case, operations with files in the `/local_vfs_dir` directory are performed locally by the `RelayVfsServer` program, and operations with files in the `/remote_vfs_dir` directory are redirected to the `VfsServer` program via IPC.

Building and running the example

The CMake system, which is included with KasperskyOS Community Edition SDK, is used to build and run the example.

See Building and running examples section.

virtiofs example

This example demonstrates support for the high-performance virtual file system Virtiofs.

Example directory in the SDK

The code of the example and build scripts are available at the following path:

```
/opt/KasperskyOS-Community-Edition-<platform>-<version>/examples/virtiofs
```

List of programs

- The example includes system programs supplied in the KasperskyOS boot image and additional system programs and drivers required for working with the emulator or a specific hardware platform on which the solution will run. For more details, refer to Architecture and startup of a KasperskyOS-based solution.

  System programs are added to the solution image automatically during the build.

- `Client` – application that mounts and uses the virtiofs file system. The program build script (the `./client/CMakeLists.txt` file) contains instructions for packaging the application into a KPA package.

Security policy description

The `security.psl` file contains a description of a security policy for a KasperskyOS-based solution and is located in the `./einit/src` directory. The declarations in the PSL file are provided with comments that explain the purpose of these declarations.

The solution security policy description in this example allows any interaction between different processes and between processes and the kernel. This policy can be used only as a stub during the early stages of a solution development so that the security module does not interfere with interactions. It would be unacceptable to apply such a policy in a real-world KasperskyOS-based solution.

Additional files

The `./einit/CMakeLists.txt` file contains instructions for building a KasperskyOS boot image.

The `./einit/src/` directory contains the configuration files for the ServiceManager system program, which is responsible for installing and running KPA packages:

- The `initial_install.json` file contains a list of KPA packages that must be installed when KasperskyOS is started for the first time.

- The `system_services.json` file contains a list of installed programs that must be started each time KasperskyOS starts.

Operating scenario

The `Client` program mounts and uses the virtiofs file system while creating a shared directory with the host machine and reading the file located in it.

Building and running the example

See Building and running examples section.

Appendices

This section provides additional information to supplement the primary text of the document.

Information about certain limits set in the system

Header files and IDL files from the KasperskyOS SDK contain constants that set limits in the system (see table below).

Constants that set limits in the system

Subsystem	Constants
`POSIX`	The constants in the `sysroot-*-kos/include/limits.h` file.
`BlobContainer`	The constants in the `sysroot-*-kos/include/kl/Env.idl(.h)` files: • `MaxArgSize (kl_EntityLauncher_MaxArgSize)` – maximum program launch parameter and environment variable size in bytes. • `MaxArgsCount (kl_EntityLauncher_MaxArgsCount)` – maximum number of launch parameters and environment variables for a program.
`CertificateStorage`	The constants in the `sysroot-*-kos/include/kl/CertificateStorage.idl(.h)` files: • `MaxNumCerts (kl_CertificateStorage_MaxNumCerts)` – maximum number of certificates in a storage. • `MaxCertSize (kl_CertificateStorage_MaxCertSize)` – maximum certificate size in bytes. • `HashSize (kl_CertificateStorage_HashSize)` – size of the certificate storage hash, in bytes.

Subsystem	Constants
Tls	The constants in the sysroot-*-kos/include/kl/CertificatePolicy.idl(.h) files: • MaxDERCertDataSize (kl_CertificatePolicy_MaxDERCertDataSize) – maximum DER certificate size in bytes. • MaxHostAddressBufferSize (kl_CertificatePolicy_MaxHostAddressBufferSize) – maximum host address buffer size in bytes. The constants in the sysroot-*-kos/include/kl/crypto/tls/TlsEvent.idl(.h) files: • FunctionNameSize (kl_crypto_tls_TlsEvent_FunctionNameSize) – maximum function name size in bytes. • IdSize (kl_crypto_tls_TlsEvent_IdSize) – session identifier size in bytes. • HostnameSize (kl_crypto_tls_TlsEvent_HostnameSize) – maximum host name size in bytes. • PkiEntrySize (kl_crypto_tls_TlsEvent_PkiEntrySize) – maximum PKI certificate size in bytes. • MaxCertificatesInChain (kl_crypto_tls_TlsEvent_MaxCertificatesInChain) – maximum number of certificates in a chain. • MaxCertificatesInTrustedSet (kl_crypto_tls_TlsEvent_MaxCertificatesInTrustedSet) – maximum number of trusted certificates. • KeyFingerprintLength (kl_crypto_tls_TlsEvent_KeyFingerprintLength) – key fingerprint size in bytes. • MbedTlsDescriptionSize (kl_crypto_tls_TlsEvent_MbedTlsDescriptionSize) – maximum MbedTLS error description size in bytes. • VfsDescriptionSize (kl_crypto_tls_TlsEvent_VfsDescriptionSize) – maximum VFS error description size in bytes. • DescriptionSize (kl_crypto_tls_TlsEvent_DescriptionSize) – maximum event description size in bytes.
ExecutionManager	The constants in the sysroot-*-kos/include/kl/execution_manager/Types.idl(.h) files: • NkAppNameMaxSize (kl_execution_manager_Types_NkAppNameMaxSize) – maximum program name size in bytes. • NkPathMaxSize (kl_execution_manager_Types_NkPathMaxSize) – maximum executable file path size in bytes. • NkEntityNameMaxSize (kl_execution_manager_Types_NkEntityNameMaxSize) – maximum process name size in bytes. • NkEiidMaxSize (kl_execution_manager_Types_NkEiidMaxSize) – maximum process class name size in bytes. • NkTaskNameMaxSize (kl_execution_manager_Types_NkTaskNameMaxSize) – maximum process name size in bytes. • NkArgMaxLen (kl_execution_manager_Types_NkArgMaxLen) – maximum program launch parameter size in bytes. • NkEnvMaxLen (kl_execution_manager_Types_NkEnvMaxLen) – maximum environment variable size in bytes. • NkArgsArrayMaxSize (kl_execution_manager_Types_NkArgsArrayMaxSize) – maximum number of program launch parameters. • NkEnvsArrayMaxSize (kl_execution_manager_Types_NkEnvsArrayMaxSize) – maximum number of environment variables for a program.

Subsystem	Constants
VFS	The constants in the `sysroot-*-kos/include/kl/VfsTypes.idl(.h)` files: • `MaxBytesCount (kl_VfsTypes_MaxBytesCount)` – maximum `VFS` data transmission buffer size in bytes. • `MaxPathSize (kl_VfsTypes_MaxPathSize)` – maximum path size in bytes. • `MaxDevnameSize (kl_VfsTypes_MaxDevnameSize)` – maximum device name size in bytes. • `MaxFstypeSize (kl_VfsTypes_MaxFstypeSize)` – maximum file system name size in bytes. • `MaxFsDataSize (kl_VfsTypes_MaxFsDataSize)` – maximum data size for the `data` parameter of the `mount()` function, in bytes. • `MaxFcntlTSize (kl_VfsTypes_MaxFcntlTSize)` – maximum data size for an optional parameter of the `fcntl()` function, in bytes. • `MaxIoctlTSize (kl_VfsTypes_MaxIoctlTSize)` – maximum data size for an optional parameter of the `ioctl()` function, in bytes. • `MaxSockAddrSize (kl_VfsTypes_MaxSockAddrSize)` – maximum IP address size in bytes. • `MaxSockOptionSize (kl_VfsTypes_MaxSockOptionSize)` – maximum data size for the `option_value` parameter of the `getsockopt()` and `setsockopt()` functions, in bytes. • `MaxHostnameSize (kl_VfsTypes_MaxHostnameSize)` – maximum host name size in bytes. • `MaxServnameSize (kl_VfsTypes_MaxServnameSize)` – maximum data size for the `servname` parameter of the `getaddrinfo()` function and the `service` parameter of the `getnameinfo()` function, in bytes. • `MaxMsgNameSize (kl_VfsTypes_MaxMsgNameSize)` – maximum data size for the `msg_name` element of the `message` parameter of the `recvmsg()` and `sendmsg()` functions, in bytes. • `MaxMsgDataSize (kl_VfsTypes_MaxMsgDataSize)` – maximum data size for a `msg_control` element of the `message` parameter of the `recvmsg()` and `sendmsg()` functions, in bytes. • `MaxIovDataSize (kl_VfsTypes_MaxIovDataSize)` – maximum number of the buffer described by the `iovec` structure in the `message` parameter of the `recvmsg()` and `sendmsg()` functions, and in the `iov` parameter of the `readv()` and `writev()` functions, in bytes. • `MaxIovecsCount (kl_VfsTypes_MaxIovecsCount)` – maximum number of `iovec` structures in the `message` parameter of the `recvmsg()` and `sendmsg()` functions, and in the `iov` parameter of the `readv()` and `writev()` functions. • `MaxAddrinfoSize (kl_VfsTypes_MaxAddrinfoSize)` – maximum data size for the `res` parameter of the `getaddrinfo()` function, in bytes. • `VfsHostent (kl_VfsTypes_MaxHostentSize)` – maximum data size for a return value of the `gethostbyname()` function, in bytes. • `VfsDnsName (kl_VfsTypes_MaxDnsNameSize)` – maximum data size for the `name` parameter of the `getnetbyname()` function, in bytes. • `MaxProtoentNameSize (kl_VfsTypes_MaxProtoentNameSize)` – maximum protocol name size in the `name` parameter of the `getprotobyname()` function, and in the return value of the `getprotobyname()` and `getprotobynumber()` functions, in bytes. • `MaxProtoentAliasesSize (kl_VfsTypes_MaxProtoentAliasesSize)` – maximum protocol alias size in the return values of the `getprotobyname()` and `getprotobynumber()` functions, in bytes.
Dhcpcd	The constants in the `sysroot-*-kos/include/kl/rump/DhcpcdConfig.idl(.h)` files: • `MaxDhcpcdStrSize (kl_rump_DhcpcdConfig_MaxDhcpcdStrSize)` – maximum size of a parameter set received from a DHCP server, in bytes.

Subsystem	Constants
`Terminal`	The constants in the `sysroot-*-kos/include/kl/Terminal.idl(.h)` files: • `MaxTerminalBytesCount (kl_Terminal_MaxTerminalBytesCount)` – maximum terminal read/write buffer size in bytes. • `MaxTerminalConnectionIdSize (kl_Terminal_MaxTerminalConnectionIdSize)` – maximum terminal identifier size in bytes.

Preparing the WiFi driver for operation in Raspberry Pi 4 B

Initialization and operation of the Wi-Fi driver on the Raspberry Pi 4 (cyw43455 chip) hardware platform requires specialized firmware. This software must be downloaded and installed on the system prior to building examples that use the Wi-Fi driver. To automate this process, the `/opt/KasperskyOS-Community-Edition-<platform>-<version>/common/get_wifi_firmware.sh` script is supplied as part of KasperskyOS Community Edition SDK.

Command syntax for calling the `get_wifi_firmware.sh` script:

```
$ sudo ./get_wifi_firmware.sh [-s <DIR>] [-h]
```

Parameters:

- `-s <DIR>`
  Root directory for downloading files. Default value: `~/.local/share/kos/`.
- `-h`
  Displays the Help text.

When the script is run, the `lib/firmware/brcm/` subdirectory will be created relative to the directory specified in the `-s` parameter in accordance with the standard file hierarchy for firmware. The following downloaded files will be placed in this subdirectory:

- Wi-Fi controller firmware (`*.bin`)
- Calibration parameters (`*.txt`)
- Power settings (`*.clm_blob`)
- End User License Agreement (`*.LICENSE.txt`)

You can use the downloaded files to develop programs that use the WiFi driver. To do so, when building, you need to explicitly specify the path to the root directory containing the downloaded files, for example, via an environment variable or build parameter.

Developing drivers for KasperskyOS

A driver in KasperskyOS can be a local process, with one process completely capturing the device, or it can be distributed across processes, with one process capturing the device and giving access to other processes. For driver development, KasperskyOS Community Edition SDK provides the `kdf` library (header files in `sysroot-*-kos/include/kdf`, in KasperskyOS Community Edition SDK).

Glossary

Address Space Layout Randomization

Address Space Layout Randomization Address Space Layout Randomization (ASLR) is the use of random addresses for the location of process memory regions occupied by an ELF image, dynamic libraries, and the stack and heap of a program to make it more difficult to exploit vulnerabilities associated with a process address space structure that is known by a hacker in advance.

Related sections:
Managing processes (low-level API task_api.h)
Managing processes (high-level API task.h)
Preparing loadable ELF image segments for creating a process (elf_api.h)
Analyzing stack backtrace data when a process abnormally terminates
Processes endpoint

Affinity mask

An *affinity mask* is a bit mask indicating which specific processors or processor cores must execute a thread.

Related sections:
Managing threads (low-level API thread_api.h)

Application

Program that is designed for interaction with a solution user and for performing user tasks.

Related sections:
Building a KasperskyOS-based solution

Arena chunk descriptor

Structure containing the offset of the arena chunk in bytes (relative to the start of the arena) and the size of the arena chunk in bytes.

Related sections:
Working with an IPC message arena

Arena descriptor

Structure containing three pointers: one pointer to the start of the arena, one pointer to the start of the unused part of the arena, and one pointer to the end of the arena.

Related sections:
Working with an IPC message arena

ASLR

Address Space Layout Randomization

Callable handle

A *callable handle* is a client IPC handle that simultaneously identifies an IPC channel to a server and an endpoint of this server.

Related sections:
Creating handles
Dynamically creating IPC channels using the DCM system program
IDL data types

Capability

Each handle is associated with access rights to the resource identified by this handle, which means it is a capability in terms of the capability-based security mechanism known as Object Capability (OCap). By receiving a handle, a process obtains the access rights to the resource that is identified by this handle. For example, these access rights may consist of read permissions, write permissions, and/or permissions to allow another process to perform operations with a resource (handle transfer permission).

Related sections:
Resource Access Control
Managing handles (handle_api.h)

CDL

Component Definition Language is a declarative language used to create a formal specification of a solution component.

Related sections:
Formal specifications of KasperskyOS-based solution components
CDL description

Client

In the context of IPC, *client* refers to the client process.

Client library of the solution component

Transport library that converts local calls into IPC requests.

Related sections:
Transport code for IPC

Client process

Process that uses the endpoint of another process via the IPC mechanism. One process can use multiple IPC channels at the same time. A process may act as a client for some IPC channels while acting as a server for other IPC channels.

Related sections:
IPC mechanism

Condition variable

Synchronization primitive that is used to notify one or more threads about the fulfillment of a condition required by these threads. It is used together with a mutex.

Related sections:
Using condition variables (condvar.h)

Constant part of an IPC message

The part of an IPC message that contains the RIID, MID and (optionally) fixed-size parameters of interface methods.

Related sections:
Overview: IPC message structure
Working with an IPC message arena
IDL data types

CPU mask

Bit mask indicating the set of processors (processor cores). A flag set in the *i*th bit indicates that a processor with the *i* index is included in the set (numbering starts at zero).

Related sections:
Power management (pm_api.h)

Critical section

Section of code in which the resources shared by threads are accessed.

Related Sections:
Using synchronization primitives (event.h, mutex.h, rwlock.h, semaphore.h, condvar.h, sync_api.h, sync_types.h, sync.h)

DCM handle

A *DCM handle* is an ID used in the API of the system program `DCM` (Dynamic Connection Manager).

Related sections:
Dynamically creating IPC channels using the DCM system program

Describing a security policy for a KasperskyOS-based solution

Set of interrelated text files with the `psl` extension that contain declarations in the PSL language.

Related sections:
Describing a security policy for a KasperskyOS-based solution
General information about a KasperskyOS-based solution security policy description
Example descriptions of basic security policies for KasperskyOS-based solutions
Methods of KasperskyOS core endpoints

Direct memory access

Direct memory access (DMA) is a feature that allows data exchange between devices and the main system memory independently of the processor.

Related sections:
Using DMA (dma.h)
Managing I/O memory isolation (iommu_api.h)

DMA

Direct memory access

DMA buffer

Buffer that consists of one or more physical memory regions (blocks) that are used for direct memory access.

Related sections:
Using DMA (dma.h)
Managing I/O memory isolation (iommu_api.h)
Allocating and freeing memory

EDL

Entity Definition Language is a declarative language used to create a formal specification of a solution component.

Related sections:
Formal specifications of KasperskyOS-based solution components
EDL description

Empty process

A process during the initial stage after its creation when the ELF image of the program is not yet loaded into process memory.

Related sections:
Managing processes (high-level API task.h)
Managing processes (low-level API task_api.h)

Endpoint

Set of logically related methods available via the IPC mechanism (for example, an endpoint for receiving and transmitting data over the network, or an endpoint for handling interrupts).

Related sections:
Overview
IPC mechanism
Methods of KasperskyOS core endpoints

Endpoint interface

Set of signatures for endpoint methods. The endpoint interface is defined in the IDL description.

Related sections:
Formal specifications of KasperskyOS-based solution components
IDL description

Endpoint method

Interface method

Endpoint method ID

An *endpoint method ID* (MID) is the sequence number of the endpoint method within the set of methods of this endpoint (starting at zero).

Related sections:
IPC mechanism
Overview: IPC message structure

Event

Synchronization primitive that is used to notify one or more threads about the fulfillment of a condition required by these threads.

Related sections:
Using events (event.h)

Event mask

Value whose bits are interpreted as events that should be tracked or that have already occurred. An event mask has a size of 32 bits and consists of a general part and a specialized part. The common part describes events that are not specific to any resources. The specialized part describes events that are specific to certain resources.

Related sections:
Using notifications (notice_api.h)
Transferring handles

Execute interface

Interface used by the KasperskyOS kernel when querying the Kaspersky Security Module to notify it about kernel startup or about initiating the startup of a process by the kernel or by other processes.

Related sections:
Setting the global parameters of a KasperskyOS-based solution security policy
Binding methods of security models to security events

Formal specification of the KasperskyOS-based solution component

A system of IDL, CDL and EDL descriptions of a solution component (IDL and CDL descriptions are optional).

Related sections:
Formal specifications of KasperskyOS-based solution components

Frozen state of a process

State of a process in which its threads were terminated as a result of an unhandled exception but its resources were not freed, which means that you can get information about this process.

Related sections:
Managing processes (low-level API task_api.h)
Managing processes (high-level API task.h)
Managing threads (high-level API thread.h)
Managing threads (low-level API thread_api.h)

Futex

Low-level synchronization primitive that supports two operations: lock a thread and add it to the futex-linked queue of locked threads, and resume execution of threads from the futex-linked queue of locked threads.

Related sections:
Using futexes (sync.h)
Synchronization endpoint

Global system information

Global system information (GSI) is a structure containing system information, such as the timer count since the KasperskyOS kernel started, the timer counts per second, the number of processors (processor cores) in active state, and processor cache data.

Related sections:
Managing processes (low-level API task_api.h)

GSI

Global system information

Handle

A *handle* is an identifier of a resource (for example, a memory area, port, network interface, or IPC channel). The handle of an IPC channel is called an IPC handle.

Related sections:
Resource Access Control
Managing handles (handle_api.h)

KasperskyOS also uses the following descriptors:

- DCM handle
- Arena descriptor
- Arena chunk descriptor

Handle dereferencing

Operation in which the client sends a handle to the server, and the server receives a pointer to the resource transfer context, the permissions mask of the sent handle, and the ancestor of the handle sent by the client and already owned by the server. Dereferencing occurs when a client that called methods for working with a resource (such as read/write or access closure) sends the server the handle that was received from this server when access to the resource was opened.

Related sections:
Managing handles (handle_api.h)
Dereferencing handles

Handle inheritance tree

Hierarchy of generated resource handles stored in the KasperskyOS kernel.

Related sections:
Managing handles (handle_api.h)

Handle permissions mask

Value whose bits are interpreted as access rights to the resource that is identified by the specific handle.

Related sections:
Resource Access Control
Handle permissions mask
Managing handles (handle_api.h)

Handle transport container

Structure consisting of the following three fields: handle field, handle permissions mask field, and the resource transfer context field. It is used to transfer handles via IPC.

Related sections:
Transferring handles
OCap usage example

Hardware interrupt

Signal sent from a device to direct the processor to immediately pause execution of the current program and instead handle an event related to this device. For example, pressing a key on the keyboard invokes a hardware interrupt that ensures the required response to this pressed key (for example, input of a character).

Related sections:
Managing interrupt processing (irq.h)

IDL

Interface Definition Language is a declarative language used to create a formal specification of a solution component.

Related sections:
Formal specifications of KasperskyOS-based solution components
IDL description

Inferior

Object used in the GDB debugger to work with a process.

Related sections:
Using the GDB debugger

Init description

An init description consists of a text file containing data in YAML format that identifies the processes and IPC channels. The init description file is normally named `product_init.yaml`.

If a KasperskyOS-based solution uses programs that for some reason cannot be packed into KPA packages, you can start these programs by using the kl.ProductRomfsLoader system program, which starts programs based on the init description. You are advised to use this approach only if it is absolutely necessary. For more details, see the examples included in the SDK.

Related sections:
Overview: product_init.yaml
Example init descriptions
Init.yaml.in template

Interface method

Subprogram that is called via IPC.

Related sections:
IPC mechanism
IDL description
Methods of KasperskyOS core endpoints

Interprocess communication

Interprocess communication (IPC) is a mechanism for interaction between different processes and between a process and the KasperskyOS kernel.

Related sections:
IPC
Initializing IPC transport for interprocess communication and managing IPC request processing (transport-kos.h, transport-kos-dispatch.h)
POSIX support limitations

IPC

Interprocess communication

IPC channel

KasperskyOS kernel object that allows processes to interact with each other by transmitting IPC messages. An IPC channel has a client side and a server side, which are identified by a client and server IPC handle, respectively.

Related sections:
IPC mechanism
Creating IPC channels

IPC handle

An *IPC handle* is a handle that identifies an IPC channel. A client IPC handle is necessary for executing a `Call()` system call. A server IPC handle is necessary for executing the `Recv()` and `Reply()` system calls. The callable handle and listener handle are IPC handles.

Related sections:
IPC mechanism
Creating handles
Creating IPC channels
Dynamically creating IPC channels using the DCM system program
Initializing IPC transport for interprocess communication and managing IPC request processing (transport-kos.h, transport-kos-dispatch.h)

IPC message

Data packet that is transmitted between different processes and between processes and the KasperskyOS kernel for IPC. An IPC message contains a constant part and an (optional) arena.

Related sections:
Overview: IPC message structure

IPC message arena

Optional part of an IPC message that contains variable-size parameters of interface methods (and/or elements of these parameters).

Related sections:
Overview: IPC message structure
Working with an IPC message arena

IPC request

IPC message sent to a server from a client.

Related sections:
IPC mechanism

IPC response

IPC message sent to a client from a server.

Related sections:
IPC mechanism

IPC transport

Wrapper for the system calls for sending and receiving IPC messages that works separately with the constant part and arena of IPC messages. IPC transport is used by the transport code on the client side and by the code for receiving and sending IPC messages on the server side.

Related sections:
Initializing IPC transport for interprocess communication and managing IPC request processing (transport-kos.h, transport-kos-dispatch.h)
Initializing and de-initializing IPC transport for querying the security module (transport-kos-security.h)

KASLR

Kernel Address Space Layout Randomization

KasperskyOS

A specialized operating system based on a separation microkernel and security monitor.

Related sections:
Overview

KasperskyOS security model

Framework for implementing security policies for solutions.

Related sections:
Describing a security policy for a KasperskyOS-based solution
KasperskyOS security models

KasperskyOS-based solution

System software (including the KasperskyOS kernel and Kaspersky Security Module) and applications integrated to work as part of a software/hardware system.

Related sections:
Overview
Architecture and startup of a KasperskyOS-based solution
Building a KasperskyOS-based solution

KasperskyOS-based solution component

Program included in a solution.

Related sections:
Overview
Formal specifications of KasperskyOS-based solution components

Kernel Address Space Layout Randomization

Kernel Address Space Layout Randomization Kernel Address Space Layout Randomization (KASLR) is the use of random addresses for the location of process memory regions occupied by executable code, data, and the stack and heap of the kernel to make it more difficult to exploit vulnerabilities associated with a process address space structure that is known by a hacker in advance.

Related sections:
Using the command line of the KasperskyOS kernel

KPA package

A *KPA package* is a file in the proprietary KPA format that serves as packaging for a program intended for installation in a KasperskyOS-based solution. It includes the following elements:

- KPA package header. Unique sequence of bytes used to identify the KPA format.
- KPA package manifest. Data structure that describes a JSON file containing detailed information about the KPA package.
- KPA package components. Aligned byte sequences with arbitrary contents. KPA package components may include executable files, libraries, text data, or any other data required for the program to work.
- KPA package index. Data structure that describes the number of KPA package components, their checksums, and their sizes.

Related sections:
Working with KPA packages
KPA package manifest

KPA package manifest

A *KPA package manifest* is a JSON file that contains the information required when installing and using a KPA package.

Related sections:
Working with KPA packages
KPA package manifest

KSM

The Kaspersky Security Module is the KasperskyOS kernel module that allows or denies interaction between different processes and between processes and the kernel, and handles queries of processes via the security interface.

Related sections:
IPC control
Resource Access Control

KSS

Kaspersky Security System technology lets you implement solution security policies. This technology prescribes the creation of formal specifications of solution components and descriptions of solution security policies using security models.

Related sections:
Overview
Developing security policies

Listener handle

A *listener handle* is a server IPC handle with extended rights that allow it to add IPC channels to the set of IPC channels identified by this handle.

Related sections:
Creating IPC channels
Creating handles
Initializing IPC transport for interprocess communication and managing IPC request processing (transport-kos.h, transport-kos-dispatch.h)

MDL

Memory descriptor list

MDL buffer

Buffer that consists of one or more physical memory regions that can be mapped to the memory of multiple processes at the same time. A memory descriptor list is used to map the MDL buffer to process memory.

Related sections:
Creating shared memory
Preparing ELF image segments to be loaded into process memory
Receiving information about memory
Allocating and freeing memory

Memory barrier

A *memory barrier* is an instruction for a compiler or processor that guarantees that memory access operations specified in source code before setting a barrier will be executed before the memory access operations specified in source code after setting a barrier.

Related sections:
Using memory barriers (barriers.h)

Memory Descriptor List

A *memory descriptor list* (MDL) is a data structure containing the addresses and sizes of physical memory regions comprising the MDL buffer.

Related sections:
Creating shared memory

Message signaled interrupt

Message signaled interrupt (MSI) is a hardware interrupt that occurs when the device accesses the interrupt controller via MMIO memory.

Related sections:
Managing interrupt processing (irq.h)

MID

Endpoint method ID

MMIO

Memory-Mapped Input-Output.

Related sections:
Memory-mapped I/O (mmio.h)

MMIO memory

Physical addresses that are mapped to registers and memory of devices. (Portions of one physical address space are used for physical memory addressing and access to registers and memory of devices.)

Related sections:
Memory-mapped I/O (mmio.h)
Allocating and freeing memory

Mutex

A synchronization primitive that provides for mutually exclusive execution of critical sections.

Related sections:
Using mutexes (mutex.h)
POSIX support limitations

Notification receiver

KasperskyOS kernel object that collects notifications about events connected with resources.

Related sections:
Using notifications (notice_api.h)
Managing handles (handle_api.h)

OCap

Object Capability is a security mechanism that is based on capabilities.

Related sections:
Resource Access Control
Managing handles (handle_api.h)

PAL

Policy Assertion Language is a declarative language used to create solution security policy tests.

For more details, refer to the pal_tests example in the SDK.

PCB

Process control block

Process

A running program that has the following distinguishing characteristics:

- It can provide endpoints to other processes and/or use the endpoints of other processes via the IPC mechanism.
- It uses KasperskyOS core endpoints via the IPC mechanism.
- It is associated with a solution security policy that regulates the interactions of the process with other processes and with the KasperskyOS kernel.

Related sections:
Overview
Starting processes
Init.yaml.in template
Managing processes (high-level API task.h)
Managing processes (low-level API task_api.h)

Process control block

A *process control block* (PCB) is a structure containing process information that is used by the KasperskyOS kernel to manage this specific process.

Related sections:
Managing processes (low-level API task_api.h)

Program

Code that is executed within the context of an individual process.

Related sections:
Building a KasperskyOS-based solution

PSL

Policy Specification Language is a declarative language used to create a solution security policy description.

Related sections:
Describing a security policy for a KasperskyOS-based solution
PSL language syntax

Read-write lock

Synchronization primitive that is used to allow access to resources shared between threads for either write access for one thread or read access for multiple threads at the same time.

Related sections:
Using read-write locks (rwlock.h)

Recursive mutex

Mutex that can be acquired by a single thread multiple times.

Related sections:
Using mutexes (mutex.h)

Resource

KasperskyOS-based software/hardware system object that can be accessed by processes. Resources can be system resources or user resources.

Related sections:
Resource Access Control

Resource consumer

Process that uses the resources provided by the KasperskyOS kernel or other processes.

Related sections:
Resource Access Control
Managing handles (handle_api.h)
Mic security model

Resource integrity level

Level of trust afforded to a resource. The level of trust in a resource depends on whether this resource was created by a trusted subject within a software/hardware system running KasperskyOS or if it was received from an untrusted external software/hardware system, for example.

Related sections:
Mic security model

Resource provider

Process that manages user resources and manages access to those resources for other processes. For example, drivers are resource providers.

Related sections:
Resource Access Control
Managing handles (handle_api.h)

Resource transfer context

Data that allows the server to identify the resource and its state when access to the resource is requested via descendants of the transferred handle. This normally consists of a data set with various types of data (structure). For example, the transfer context of a file may include the name, path, and cursor position.

Related sections:
Managing handles (handle_api.h)

Resource transfer context object

KasperskyOS kernel object that stores the pointer to the resource transfer context.

Related sections:
Managing handles (handle_api.h)

RIID

Runtime Implementation Identifier

Runtime Implementation Identifier

A *Runtime Implementation Identifier* (RIID) is the sequence number of an endpoint within the set of endpoints of a server (starting at zero).

Related sections:
IPC mechanism
Overview: IPC message structure

SCP

Static connection page

Security audit

The Kaspersky Security Module sends information about its decisions (security audit data) to the logging system, which outputs this data (for example, via the UART) or saves it to a file.

Related sections:
Creating security audit profiles
Examples of security audit profiles

Security audit configuration

Element of a security audit profile that defines the security model objects covered by the security audit and the conditions for performing the security audit.

Related sections:
Creating security audit profiles
Examples of security audit profiles

Security audit data

Information about decisions made by the Kaspersky Security Module, including the actual decisions ("granted" or "denied"), descriptions of security events, results from calling methods of security models, and data on incorrect IPC messages.

Related sections:
Creating security audit profiles

Security audit profile

Set of security audit configurations, each of which defines the security model objects covered by the security audit and the conditions for performing the security audit.

Related sections:
Creating security audit profiles
Binding methods of security models to security events
Examples of security audit profiles
Setting the global parameters of a KasperskyOS-based solution security policy

Security audit runtime-level

The security audit runtime-level is a global parameter of a solution security policy and consists of an unsigned integer that defines the active security audit configuration. (The word "level" here refers to the configuration variant and does not necessarily involve a hierarchy.)

Related sections:
Creating security audit profiles
Setting the global parameters of a KasperskyOS-based solution security policy

Security context

Data that is associated with a security ID and is used by the Kaspersky Security Module to make decisions.

Related sections:
Resource Access Control

Security event

A signal indicating the initiation of communication between a process and another process or between a process and the KasperskyOS kernel.

Related sections:
General information about a KasperskyOS-based solution security policy description
Binding methods of security models to security events
Examples of binding security model methods to security events

Security ID

A *security identifier* (SID) is a globally unique identifier of a resource. The Kaspersky Security Module identifies resources based on their security IDs.

Related sections:
Resource Access Control
Getting a security ID (SID)

Security interface

Interface that is used for interaction between a process and the Kaspersky Security Module. The security interface is defined in the IDL description.

Related sections:
Formal specifications of KaspersкyOS-based solution components
EDL description
CDL description
IDL description
Binding methods of security models to security events
Initializing and de-initializing IPC transport for querying the security module (transport-kos-security.h)

Security model expression

Security model method that returns values that can be used as input data for other methods of security models.

Related sections:
General information about a KaspersкyOS-based solution security policy description
Binding methods of security models to security events
KaspersкyOS security models

Security model method

Element of a security model that determines the permissibility of interactions between various processes and between processes and the KaspersкyOS kernel.

Related sections:
General information about a KaspersкyOS-based solution security policy description
Binding methods of security models to security events
Examples of binding security model methods to security events
KaspersкyOS security models

Security model object

Instance of a class whose definition is a formal description of a security model (in a PSL file).

Related sections:
General information about a KaspersкyOS-based solution security policy description
Creating security model objects
KaspersкyOS security models

Security model rule

Security model method that returns a "granted" or "denied" decision.

Related sections:
General information about a KaspersкyOS-based solution security policy description
Binding methods of security models to security events
Examples of binding security model methods to security events
KaspersкyOS security models

Security module decision

A decision on whether to allow or deny a specific interaction between different processes or between a process and the KasperskyOS kernel.

Related sections:
Overview
IPC control
General information about a KasperskyOS-based solution security policy description

Security pattern

A security pattern (or security template) describes a specific recurring security issue that arises in certain known contexts, and provides a well-proven, general scheme for resolving this kind of security issue. A pattern is not a finished project that can be converted directly into code. Instead, it is a solution to a general problem encountered in various projects.

Related sections:
Security patterns for development under KasperskyOS

Security pattern system

Set of security patterns together with instructions on their implementation, combination, and practical use when designing secure software systems.

Related sections:
Security patterns for development under KasperskyOS

Security policy for a KasperskyOS-based solution

Logic for processing security events in the solution. This logic is implemented by the Kaspersky Security Module. The source code of the Kaspersky Security Module is generated from the solution security policy description and formal specifications of solution components.

Related sections:
Overview

Security template

A security template (or security pattern) describes a specific recurring security issue that arises in certain known contexts, and provides a well-proven, general scheme for resolving this kind of security issue. A template is not a finished project that can be converted directly into code. Instead, it is a solution to a general problem encountered in various projects.

Related sections:
Security patterns for development under KasperskyOS

Seed

Starting number of the random number generator (seed) , which determines the sequence of the generated random numbers. In other words, if the same seed value is set, the generator creates identical sequences of random numbers. (The entropy of these numbers is fully determined by the entropy of the seed value, which means that these numbers are not entirely random, but pseudorandom.)

Related sections:
Generating random numbers (random_api.h)
Managing processes (low-level API task_api.h)

Semaphore

Synchronization primitive based on a counter whose value can be atomically changed.

Related sections:
Using semaphores (semaphore.h)
POSIX support limitations

Server

In the context of IPC, *server* refers to the server process.

Server library of the solution component

Transport library that converts IPC requests into local calls.

Related sections:
Transport code for IPC

Server process

Process that provides endpoints to other processes via the IPC mechanism. One process can use multiple IPC channels at the same time. A process may act as a server for some IPC channels while acting as a client for other IPC channels.

Related sections:
IPC mechanism

SID

Security ID

SMP

Symmetric Multiprocessing

Related sections:
Using the GDB debugger

Static connection page

A *static connection page* (SCP) is a set of structures containing data for creating IPC channels, startup parameters, and environment variables of a program.

Related sections:
Managing processes (low-level API task_api.h)

Subject integrity level

Level of trust afforded to a subject. The trust level of a subject depends on whether the subject interacts with untrusted external software/hardware systems or whether it has a proven standard of quality, for example.

Related sections:
Mic security model

System program

A program that creates the infrastructure for application software (for example, it facilitates hardware operations, supports the IPC mechanism, and implements file systems and network protocols).

Related sections:
Building a KasperskyOS-based solution

System resource

Resource that is managed by the KasperskyOS kernel. Some examples of system resources include processes, memory regions, and interrupts.

Related sections:
Resource Access Control
Managing handles (handle_api.h)

TCB

Thread control block

Thread

A *thread* is an abstraction used to manage the execution of program code. One process can include one or more threads. CPU time is allocated separately for each thread. Each thread may execute the entire code of the program or just a part of the code. The same program code may be executed in multiple threads.

Related sections:
Managing threads (low-level API thread_api.h)
Managing threads (high-level API thread.h)
POSIX support limitations

Thread control block

A *thread control block* (TCB) is a structure containing thread information that is used by the KasperskyOS kernel to manage this specific thread.

Related sections:
Managing threads (low-level API thread_api.h)
Managing threads (high-level API thread.h)

Thread identifier

Thread identifier (TID) is an integer value that is unique for each thread within an entire solution. It is assigned by the KasperskyOS kernel. TIDs are not used to manage threads but instead are used to identify them – for example, in certain APIs of the `libkos` library and when debugging and logging programs.

Related sections:
Managing threads (high-level API thread.h)
Managing threads (low-level API thread_api.h)
Managing processes (low-level API task_api.h)
Getting values of performance counters (value_api.h)
Logging the operation of programs (trace.h)
Debugging programs in a KasperskyOS-based solution

Thread local storage

A *thread local storage* (TLS) is the process memory in which a thread can store data in isolation from other threads.

Related sections:
Managing threads (high-level API thread.h)
Managing threads (low-level API thread_api.h)

TID

Thread identifier

TLS

Thread local storage

Transport code

Methods and types in C or C++ for conducting IPC.

Related sections:
Transport code for IPC
Example generation of transport methods and types
Transport code in C++
Creating EDL files and generating transport code in C
Generating transport code in C++

Transport library

To use a supplied solution component via IPC, the KasperskyOS SDK provides the following transport libraries:

- Solution component's client library, which converts local calls into IPC requests.
- Solution component's server library, which converts IPC requests into local calls.

Related sections:
Transport code for IPC

User resource

Resource that is managed by a process. Examples of user resources: files, input-output devices, data storage.

Related sections:
Resource Access Control
Managing handles (handle_api.h)

User resource context

Data that allows the resource provider to identify the resource and its state when access to the resource is requested by other processes. This normally consists of a data set with various types of data (structure). For example, the context of a file may include the name, path, and cursor position.

Related sections:
Managing handles (handle_api.h)

Working set

Working set (WS) is an abstraction that describes the use of physical memory in a system. This is a KasperskyOS kernel object associated with data such as the total number of physical memory pages in the system, the number of occupied physical memory pages in the system, and threshold values for the amount of occupied physical memory in the system.

Related sections:
Using a working set (ws_api.h)
Physical memory endpoint

WS

Working set

Zero page deduplication

Zero Page Deduplication (ZPD) is a memory management mechanism that, at a specified frequency, finds and frees pages of physical memory that are completely filled with zeros, and also maps the pages of virtual memory corresponding to the freed pages of physical memory to a single shared page of physical memory that is completely filled with zeros.

Related sections:
Using the KasperskyOS kernel command line
Obtaining statistical data on the system through the libkos library API

ZPD

Zero page deduplication

Legal notice

This section provides legal information.

Licensing

The *End User License Agreement* is a legally binding agreement between you and AO Kaspersky that stipulates the terms on which you may use KasperskyOS Community Edition SDK.

> Please carefully read the terms of the End User License Agreement before you begin using KasperskyOS Community Edition SDK.

You can view the terms of the End User License Agreement (EULA) in the following ways:

- Read the text of the End User License Agreement before downloading the KasperskyOS Community Edition SDK distribution package.
- Read the document named EULA.<language code>.txt located in the directory `/opt/KasperskyOS-Community-Edition-<platform>-<version>` after installing KasperskyOS Community Edition SDK.

You accept the terms of the End User License Agreement by selecting the **I agree** check box under the text of the End User License Agreement before downloading the KasperskyOS Community Edition SDK distribution package.

If you do not accept the terms of the End User License Agreement, you must cancel the download of the distribution package and must not use KasperskyOS Community Edition SDK.

Data provision

KasperskyOS Community Edition SDK versions

KasperskyOS Community Edition SDK is distributed in two different versions:

- Version that can be downloaded from the Russian-language website https://os.kaspersky.ru/development.
- Version that can be downloaded from the English-language website https://os.kaspersky.com/development.

Versions differ in the contents of their End User License Agreement, file and differ in the specific information that they automatically transmit to Kaspersky servers when a solution is built using CMake libraries from the SDK.

Data provision in KasperskyOS Community Edition SDK

The version of KasperskyOS Community Edition SDK downloaded from the Russian-language website automatically transmits the following information to Kaspersky servers when a solution build is started:

- Number of the installed version of KasperskyOS Community Edition SDK.
- Unique hardware ID consisting of the checksum of the creation date of the directory /opt/KasperskyOS-Community-Edition-<platform>-<version>.

The version of KasperskyOS Community Edition SDK downloaded from the English-language website automatically transmits the following information to Kaspersky servers when a solution build is started:

- Number of the installed version of KasperskyOS Community Edition SDK.

In addition to data transmission during the solution build process, both versions also check for the availability of a newer version of KasperskyOS Community Edition SDK.

> If there is no active Internet connection, the solution build occurs without transmitting data or checking for updates.

You can disable the check for SDK updates and transmission of SDK version information to the Kaspersky server by using the `NO_NEW_VERSION_CHECK` parameter of the CMake command `initialize_platform()` during the solution build.

Data is transmitted to account for the number of users of KasperskyOS Community Edition SDK and to obtain information about the distribution and use of KasperskyOS Community Edition SDK.

Any received information is protected by Kaspersky in accordance with the requirements established by law and in accordance with current Kaspersky regulations. Data is transmitted over encrypted communication channels.

Information about third-party code

Information about third-party code is contained in the file named legal_notices.txt in the application installation folder.

Trademark notices

Registered trademarks and service marks are the property of their respective owners.

Adobe, PostScript are either registered trademarks or trademarks of Adobe in the United States and/or other countries.

Amazon, AWS are trademarks of Amazon.com, Inc. or its affiliates.

AFS, AIX, AIX 5L, Anynet, Approach, AS/400, Blue Gene, BookManager, BNT, C/VM, DB2, DYNIX/ptx, Domino, DXT, ESCON, eServer, IA, IBM, IMS, iSeries, ibm.com, Language Environment, MVS, POWER7, POWER8, PowerPC, RDN, Rhapsody, RS/6000, s3, S/390, OS/400, Solid, System z, System/390, Verilog, VisualAge, QMF, RDN, z9, z10, z/Architecture, z/OS, and z/VM are trademarks of International Business Machines Corporation, registered in many jurisdictions worldwide.

Aironet, Cisco, IOS, and Jabber are registered trademarks or trademarks of Cisco Systems, Inc. and/or its affiliates in the United States and certain other countries.

AMD, AMD64 are trademarks or registered trademarks of Advanced Micro Devices, Inc.

Android, Chrome, Closure, Dalvik, Dart, DoubleClick, Google, Google Chrome, Google Code, Google App Engine, Flutter, Gmail, Chromium, Chromebook, Google Code, Google Test, PROTOBUF, and Motif are trademarks of Google LLC.

Apache is either a registered trademark or trademark of the Apache Software Foundation in the United States and/or other countries.

Arm and Mbed are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

Aperture, Apple, AppleScript, AppleTalk, Apple TV, App Store, Aqua, Carbon, Cocoa, eMac, Face ID, Finder, FireWire, iBook, iCal, iMessage, iPhone, iWeb, Mac, MacBook, Macintosh, macOS, Mac OS, MPW, LocalTalk, Leopard, NetInfo, Newton, Objective-C, OpenCL, OS X, Power Mac, QuickDraw, QuickTime, Rosetta, Safari, Skia, Sherlock, Tiger, and Xcode are trademarks of Apple Inc.

ActiveX, AppLocker, Arc, Convergence, ClearType, Direct3D, DirectX, DirectInput, Excel, Express, Internet Explorer, IntelliSense, JScript, MapPoint, Microsoft, Microsoft Edge, MSDN, MS-DOS, MSN, MultiPoint, PowerShell, OpenType, SQL Server, Visual C++, Visual Studio, Win32, Windows, Windows Mobile, Windows Phone, Windows Server, Windows Store, Windows Vista, and Xbox are trademarks of the Microsoft group of companies.

Atom, Celeron, Core, i960, Intel, Intel386, Intel XScale, Itanium, MCS, MMX, Pentium, SpeedStep, Wireless MMX, and Xeon XMM are trademarks of Intel Corporation or its subsidiaries.

BACnet is a registered trademark of the American Society of Heating, Refrigerating and Air-Conditioning Engineers, Inc.

Dell Technologies, Dell, EMC, Isilon, OneFS, and other trademarks are trademarks of Dell Inc. or its subsidiaries.

DICOM is the registered trademark of the National Electrical Manufacturers Association for its Standards publications relating to digital communications of medical information.

Docker and the Docker logo are trademarks or registered trademarks of Docker, Inc. in the United States and/or other countries. Docker, Inc. and other parties may also have trademark rights in other terms used herein.

Eclipse Mosquitto is a trademark of Eclipse Foundation, Inc.

F5 is a trademark of F5 Networks, Inc. in the U.S. and in certain other countries.

Huawei is a trademark of Huawei Technologies Co., Ltd.

Linux is the registered trademark of Linus Torvalds in the U.S. and other countries.

OpenSSL is a trademark owned by the OpenSSL Software Foundation.

Python is a trademark or registered trademark of the Python Software Foundation.

Raspberry Pi is a trademark of the Raspberry Pi Foundation.

Fedora, Red Hat, and CentOS are trademarks or registered trademarks of Red Hat, Inc. or its subsidiaries in the United States and other countries.

Firebird is a registered trademark of the Firebird Foundation.

Firefox, Mozilla are trademarks of the Mozilla Foundation in the U.S. and other countries.

FreeBSD is a registered trademark of The FreeBSD Foundation.

Symantec, K9, PGP are trademarks or registered trademarks of Symantec Corporation or its affiliates in the U.S. and other countries.

MIPS is a trademark or registered trademark of MIPS Technologies, Inc. in the United States and other countries.

NetWare is a registered trademark of Novell Inc. in the United States and other countries.

Novell is a registered trademark of Novell Enterprises Inc. in the United States and other countries.

Nvidia is a registered trademark of NVIDIA Corporation.

Oracle, Java, and JavaScript are registered trademarks of Oracle and/or its affiliates.

QT is a trademark or registered trademark of The Qt Company Ltd.

Samsung is a trademark of SAMSUNG in the United States or other countries.

SPL is a trademark and registered trademark of Splunk Inc. in the United States and other countries.

S60 is a trademark or registered trademark of Nokia Corporation.

Texas Instruments is a trademark of Texas Instruments.

Trend Micro is a trademark or registered trademark of Trend Micro Incorporated.

Ubuntu is a registered trademark of Canonical Ltd.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

VxWorks is a registered trademark (®) or service mark (SM) of Wind River Systems, Inc. This product is not affiliated with, endorsed, sponsored, supported by the owners of the third-party trademark mentioned herein.