kaspersky

KasperskyOS Community Edition 1.2

Содержание

KSS

MID

OCap

OPP

PAL

PSL

RIID

SID

Аппаратное прерывание

Арена IPC-сообщения

Аудит безопасности

Барьер памяти

Блокировка чтения-записи

Буфер DMA

Выражение модели безопасности

Данные аудита безопасности

Дерево наследования дескрипторов

Дескриптор

Дескриптор арены

Дескриптор участка арены

Идентификатор безопасности

Идентификатор метода службы

Идентификатор службы

Инициализирующая программа

Интерфейс безопасности

Интерфейс службы

Интерфейсный метод

Клиент

Клиентская библиотека компонента решения

Клиентский процесс

Компонент решения на базе KasperskyOS

Контекст безопасности

Контекст передачи ресурса

Контекст пользовательского ресурса

Конфигурация аудита безопасности

Критическая секция

Мандатная ссылка

Маска прав дескриптора

Маска событий

Межпроцессное взаимодействие

Метод модели безопасности

Метод службы

Модель безопасности KasperskyOS

Мьютекс

Начальное значение генератора случайных чисел

Объект контекста передачи ресурса

Объект модели безопасности

Описание политики безопасности решения на базе KasperskyOS

Что нового

В KasperskyOS Community Edition 1.2 появились следующие возможности и доработки:

> В связи с изменениями в компонентах SDK, вам необходимо внести изменения в прикладной код, разработанный с использованием версии KasperskyOS Community Edition 1.1.1, перед тем как использовать его с версией KasperskyOS Community Edition 1.2. Подробнее см. "Миграция прикладного кода с версии 1.1.1 на версию 1.2".

- Изменены системные требования: для установки SDK требуется ОС Ubuntu GNU/Linux 22.04 "Jammy Jellyfish".
- Добавлена возможность использовать динамические библиотеки.
- Добавлена возможность использовать аппаратный сторожевой таймер (watchdog) на Raspberry Pi 4 Model B.
- Добавлен компонент ExecutionManager, предназначенный для создания, запуска и остановки процессов.
- Добавлен скрипт для автоматической установки переменных окружения, используемых инструментами SDK.
- Добавлена передача данных на серверы "Лаборатории Касперского" при запуске сборки примеров из состава SDK. Данные передаются с целью учета количества пользователей KasperskyOS Community Edition и получения информации о распространении и использовании KasperskyOS Community Edition. Вы можете отключить эту функциональность.
- Обновлено руководство разработчика, в частности:
  - Добавлен раздел "Работа с ареной IPC-сообщений".
  - Добавлен раздел "Сведения о некоторых лимитах, установленных в системе".
  - Добавлены описания сценариев работы с интерфейсами библиотеки libkos.
  - Обновлена инструкция по сборке и выполнению тестов политики безопасности решения.
  - Добавлен глоссарий.
- Добавлены следующие сторонние библиотеки и приложения:
  - Guidelines Support Library (GSL) (2.1.0);
  - json_scheme_validator (2.1.0);
  - libpcap (1.10.4);
  - libunwind (1.6.2);
- Обновлены следующие сторонние библиотеки и приложения:
  - libxml2;

    - Mbedtls;
    - Mosquitto;
    - OpenSSL;
    - spdlog;
    - sqlite;
    - fmt;
    - zlib
    - flex;
    - bison;
    - QEMU.
- Исключены из состава SDK следующие сторонние библиотеки и приложения:
    - ffmpeg;
    - opencv;
    - libjpeg-turbo;
    - libpng;
    - protobuf.

В KasperskyOS Community Edition 1.1.1 появились следующие возможности и доработки:

- Обновлены следующие сторонние библиотеки и приложения:
    - FFmpeg;
    - libxml2;
    - Eclipse Mosquitto;
    - opencv;
    - OpenSSL;
    - protobuf;
    - sqlite;
    - usb.
- Добавлена поддержка аппаратной платформы Raspberry Pi 4 Model B ревизии 1.5.

В KasperskyOS Community Edition 1.1 появились следующие возможности и доработки:

- Добавлена поддержка работы с шиной I2C в режиме ведущего устройства (master).
- Добавлена поддержка работы с шиной SPI в режиме ведущего устройства (master).
- Добавлена поддержка для USB HID устройств.
- Добавлена поддержка симметричной многопроцессорности (SMP).
- Расширены возможности для профилирования устройства: добавлена библиотека iperf и счетчики, отслеживающие системные параметры.
- Добавлена библиотека PCRE и пример работы с ней.
- Добавлена библиотека SPDLOG и пример работы с ней.
- Добавлен компонент MessageBus и пример работы с ним.
- Добавлены средства динамического анализа кода (ASAN, UBSAN).

В KasperskyOS Community Edition 1.0 появились следующие возможности и доработки:

- Добавлена поддержка аппаратной платформы Raspberry Pi 4 Model B.
- Добавлена поддержка SD-карты для аппаратной платформы Raspberry Pi 4 Model B.
- Добавлена поддержка Ethernet для аппаратной платформы Raspberry Pi 4 Model B.
- Добавлена поддержка портов ввода-вывода GPIO для аппаратной платформы Raspberry Pi 4 Model B.
- Добавлены сетевые сервисы DHCP, DNS, NTP и примеры работы с ними.
- Добавлена библиотека для работы с протоколом MQTT и примеры ее использования.

О KasperskyOS Community Edition

KasperskyOS Community Edition (CE) — общедоступная версия KasperskyOS, предназначенная для освоения основных принципов разработки приложений под KasperskyOS. KasperskyOS Community Edition позволит вам увидеть, как концепции, заложенные в KasperskyOS, работают на практике. KasperskyOS Community Edition включает в себя примеры приложений с исходным кодом, подробные пояснения, а также инструкции и инструменты для сборки приложений.

KasperskyOS Community Edition пригодится вам для:

- изучения принципов и приемов разработки "secure by design" на практических примерах;
- изучения KasperskyOS как возможной платформы для реализации своих проектов;
- прототипирования решений (прежде всего, Embedded/IoT) на основе KasperskyOS;
- портирования приложений/компонентов на KasperskyOS;
- изучения вопросов безопасности в разработке ПО.

KasperskyOS Community Edition позволяет разрабатывать приложения как на языке C, так и на C++. Подробнее о настройке среды разработки см. "Настройка среды разработки".

Для получения KasperskyOS Community Edition перейдите по ссылке.

Помимо этой документации, также рекомендуем изучить материалы раздела сайта KasperskyOS для разработчиков.

Об этом документе

Руководство разработчика KasperskyOS Community Edition адресовано специалистам, которые осуществляют разработку безопасных решений на базе KasperskyOS.

Руководство предназначено специалистам, обладающим следующими навыками: знание языков C/C++, опыт разработки под POSIX-совместимые системы, знакомство с GNU Binary Utilities (далее также "binutils").

Вы можете применять информацию в этом руководстве для выполнения следующих задач:

- установка и удаление KasperskyOS Community Edition;
- использование KasperskyOS Community Edition.

Комплект поставки

KasperskyOS SDK представляет собой набор программных средств для создания решений на базе KasperskyOS.

В комплект поставки KasperskyOS Community Edition входят:

- deb-пакет для установки KasperskyOS Community Edition, содержащий:
  - образ ядра операционной системы KasperskyOS;

- инструменты для разработки (компилятор GCC, компоновщик LD, набор утилит binutils, эмулятор QEMU и сопутствующие инструменты);
- утилиты и скрипты (например, генераторы исходного кода, скрипт `makekss` для создания модуля безопасности Kaspersky Security Module, скрипт `makeimg` для создания образа решения);
- набор библиотек, обеспечивающих частичную совместимость со стандартом POSIX;
- драйверы;
- системные программы (например, виртуальную файловую систему);
- примеры работы с компонентами KasperskyOS Community Edition;
- лицензионное соглашение;
- файл с информацией о стороннем коде (Legal Notices).

- Руководство разработчика KasperskyOS Community Edition (онлайн-документация).
- Информация о версии (Release Notes);

KasperskyOS SDK устанавливается на компьютер под управлением ОС Ubuntu GNU/Linux®.

Следующие компоненты, входящие в комплект поставки KasperskyOS Community Edition, являются "Runtime компонентами" в соответствии с условиями лицензионного соглашения:

- Образ ядра операционной системы KasperskyOS.

Остальные части комплекта поставки не являются "Runtime компонентами". Условия и возможности использования каждого компонента могут быть дополнительно указаны в разделе "Информация о стороннем коде".

Системные требования

Для установки KasperskyOS Community Edition и запуска примеров под QEMU необходимы:

1. **Операционная система:** Ubuntu GNU/Linux 22.04 "Jammy Jellyfish". Возможно использование Docker-контейнера.
2. **Процессор:** процессор с архитектурой x86-64 (для большей производительности требуется поддержка аппаратной виртуализации).
3. **Оперативная память:** для комфортной работы с инструментами сборки рекомендуется иметь не менее 4 ГБ оперативной памяти.
4. **Дисковое пространство:** не менее 3 ГБ свободного пространства в директории `/opt` (в зависимости от разрабатываемого решения).

Для запуска примеров на аппаратной платформе Raspberry Pi необходимы:

- модель Raspberry Pi 4 Model B (ревизии 1.1, 1.2, 1.4, 1.5) с объемом оперативной памяти равным 2, 4 или 8 ГБ;
- microSD-карта объемом не менее 2 ГБ;

- преобразователь USB-UART.

Включенные сторонние библиотеки и приложения

Для упрощения процесса разработки приложений в состав KasperskyOS Community Edition также включены следующие сторонние библиотеки и приложения:

- **flex (v.2.6.2)** – генератор лексических анализаторов.

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

- **pkg-config-lite (v.0.28)** – утилита, предоставляющая интерфейс для получения информации об установленных в системе библиотеках (версия, параметры для C / C ++ компилятора и компоновщика).

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

- **CMake (v.3.25.0)** – кроссплатформенное программное средство автоматизации сборки программного обеспечения из исходного кода.

  Документация: https://cmake.org/documentation

- **autoconf-archive (v.2022.09.03)** – набор макросов для утилиты Autoconf, создающей конфигурационные скрипты для автоматической настройки и сборки программного обеспечения из исходного кода.

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

- **Automake (v.1.13 и v.1.16.4)** – утилита генерации стандартизированных файлов `Makefile.in` для автоматической настройки и сборки программного обеспечения из исходного кода.

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

- **Autoconf (v.2.69)** – утилита генерации конфигурационных скриптов `configure` для автоматической настройки и сборки программного обеспечения из исходного кода.

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

- **autotools-wrappers (v.am-10)** – обертка для утилит Autoconf и Automake, которая определяет подходящую для автоматической настройки и сборки программного обеспечения версию утилиты из нескольких, установленных в системе.

  Документация: https://gitweb.gentoo.org/proj/autotools-wrappers.git/tree

- **Libtool (v.2.4.2)** – скрипт поддержки общих библиотек, который скрывает сложности использования библиотек за последовательным, переносимым интерфейсом.

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

- **Binutils (v.2.38)** – набор утилит для работы с бинарными файлами, который включает в себя ассемблер, компоновщик, архиватор и другие утилиты.

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

- **Bison (v.3.5.4)** – генератор синтаксических анализаторов общего назначения, преобразующий аннотированную контекстно-свободную грамматику в LR- или GLR-анализатор с использованием таблиц разбора LALR(1).

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

- **GNU Compiler Collection (GCC) (v.9.2.1)** – набор компиляторов для различных языков программирования, включая C / C ++.

Документация: https://gcc.gnu.org/onlinedocs

- **QEMU (v.8.1.3)** – программа для эмуляции аппаратного обеспечения различных платформ.

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

- **Automated Testing Framework (ATF) (v.0.20)** – набор библиотек для написания тестов для программ на C, C++ и POSIX shell.

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

- **Boost (v.1.78.0)** – собрание библиотек классов, использующих функциональность языка C++ и предоставляющих удобный кроссплатформенный высокоуровневый интерфейс для лаконичного кодирования различных повседневных подзадач программирования (работа с данными, алгоритмами, файлами, потоками и т. п.).

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

- **nlohmann_json (v.3.9.1)** – библиотека для работы с форматом JSON.

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

- **Civetweb (v.1.11)** – простой в использовании, мощный, встраиваемый веб-сервер на C / C ++ с дополнительной поддержкой CGI, SSL и Lua.

  Документация: http://civetweb.github.io/civetweb/UserManual.html

- **fmt (v.9.1.0)** – библиотека для форматирования с открытым исходным кодом.

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

- **Guidelines Support Library (GSL) (v.2.1.0)** – библиотека, содержащая функции и типы, которые предлагаются к использованию в соответствии с C++ Core Guidelines при поддержке Standard C++ Foundation.

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

- **GoogleTest (v.1.10.0)** – библиотека для тестирования кода на C++.

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

- **iperf (v.3.10.1)** – библиотека для тестирования производительности сети.

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

- **json-schema-validator (v.2.1.0)** – библиотека, предназначенная для проведения валидации данных в формате JSON в соответствии с заданными JSON-схемами.

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

- **libffi (v.3.2.1)** – библиотека, предоставляющая C-интерфейс для вызова заранее скомпилированного кода.

  Документация: https://github.com/libffi/libffi

- **jsoncpp (v.1.9.4)** – библиотека для работы с форматом JSON.

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

- **libpcap (v.1.10.4)** – библиотека для разработки программ, которые могут захватывать, фильтровать и анализировать сетевой трафик в UNIX-подобных системах.

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

- **libunwind (v.1.6.2)** – библиотека для обработки исключительных ситуаций и реализации механизма обратной трассировки стека вызова функций при аварийном завершении процесса.

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

- **libxml2 (v.2.10.4)** – библиотека для работы с XML.

  Документация: http://xmlsoft.org

- **Mbed TLS (v.3.3.0)** – библиотека, предоставляющая реализацию криптографических протоколов, таких как TLS/SSL, DTLS, а также алгоритмы шифрования, хэширования и аутентификации.

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

- **Eclipse Mosquitto (v.2.0.18)** – брокер сообщений, реализующий протокол MQTT.

  Документация: https://mosquitto.org/documentation

- **NTP (v.4.2.8P15)** – библиотека для работы протоколом времени NTP.

  Документация: http://www.ntp.org/documentation.html

- **OpenSSL (v.1.1.1t)** – полноценная криптографическая библиотека с открытым исходным кодом.

  Документация: https://www.openssl.org/docs/

- **pcre (v.8.44)** – библиотека для работы с регулярными выражениями.

  Документация: https://www.pcre.org/current/doc/html

- **spdlog (v.1.11.0)** – библиотека для журналирования.

  Документация: https://github.com/gabime/spdlog

- **sqlite (v.3.41.2)** – библиотека для работы с базами данных.

  Документация: https://www.sqlite.org/docs.html

- **Zlib (v.1.2.13)** – библиотека для сжатия данных.

  Документация: https://zlib.net/manual.html

- **usb (v.13.0.0)** – библиотека для работы с USB-устройствами.

  Документация: https://github.com/freebsd/freebsd-src/tree/release/13.0.0/sys/dev/usb

- **libevdev (v.1.6.0)** – библиотека для работы с периферийными устройствами типа evdev.

  Документация: https://www.freedesktop.org/software/libevdev/doc/latest

- **dhcpcd (v.9.4.1)** – DHCP-, DHCPv6-клиент, предназначенный для автоматической конфигурации сетевых параметров на клиентской стороне.

  Документация: https://github.com/NetworkConfiguration/dhcpcd

- **Lwext4 (v.1.0.0)** – библиотека для работы с файловыми системами ext2/3/4.

  Документация: https://github.com/gkostka/lwext4.git

Также см. Информация о стороннем коде.

Ограничения и известные проблемы

Поскольку KasperskyOS Community Edition предназначен для обучения, мы интегрировали в пакет ряд ограничений:

1. Максимальное поддерживаемое количество запущенных программ: 32.

2. При завершении работы программы любым способом (например, return из основного потока исполнения) выделенные программой ресурсы не освобождаются, а сама программа переводится в "спящее" состояние. Программы не могут быть запущены повторно.

3. Не поддерживается запуск двух и более программ с одинаковым EDL-описанием.

4. Система останавливается, если не осталось работающих программ или если один из потоков программы-драйвера завершился (штатным или нештатным образом).

5. При запуске примеров на аппаратной платформе Raspberry Pi 4 Model B максимальный размер образа решения (файла kos-image) не должен превышать 248 MB.

Миграция прикладного кода с версии SDK 1.1.1 на версию SDK 1.2

В связи с изменениями в компонентах SDK в версии 1.2, вам необходимо внести изменения в прикладной код, разработанный с использованием версии KasperskyOS Community Edition 1.1.1, перед тем как использовать его с версией KasperskyOS Community Edition 1.2.

Список необходимых изменений:

1. В SDK добавился драйвер для работы с сопроцессором VideoCore (VC6) через технологию mailbox: `kl.drivers.Bcm2711MboxArmToVc`, доступ к которому необходим драйверам `kl.drivers.DNetSrv` и `kl.drivers.USB`.

   - Если для создания init-описания решения (файл `init.yaml`) используется шаблон `init.yaml.in` и для процессов `kl.drivers.DNetSrv` и `kl.drivers.USB` использованы макросы `@INIT_ProgramName_ENTITY_CONNECTIONS+@` или `@INIT_ProgramName_ENTITY_CONNECTIONS@`, то изменений в init-описании не требуется.

     Иначе, если IPC-каналы для процессов `kl.drivers.DNetSrv` и `kl.drivers.USB` указаны вручную, то необходимо добавить процесс `kl.drivers.Bcm2711MboxArmToVc` в init-описание и определить IPC-каналы между ним и процессами `kl.drivers.DNetSrv` и `kl.drivers.USB`:

```
- name: kl.drivers.Bcm2711MboxArmToVc
  path: bcm2711_mbox_arm2vc_h

- name: kl.drivers.USB
  path: usb
  connections:
  ...
  - target: kl.drivers.Bcm2711MboxArmToVc
    id: kl.drivers.Bcm2711MboxArmToVc

- name: kl.drivers.DNetSrv
  path: dnet_entity
  connections:
  ...
  - target: kl.drivers.Bcm2711MboxArmToVc
    id: kl.drivers.Bcm2711MboxArmToVc
```

   - Необходимо добавить процесс `kl.drivers.Bcm2711MboxArmToVc` в файл `security.psl` и разрешить процессам `kl.drivers.DNetSrv` и `kl.drivers.USB` и ядру взаимодействовать с ним:

```
...
use kl.drivers.Bcm2711MboxArmToVc._

...

execute src = Einit dst = kl.drivers.Bcm2711MboxArmToVc  { grant () }

request src = kl.drivers.Bcm2711MboxArmToVc dst = kl.core.Core  { grant () }
response src = kl.core.Core dst = kl.drivers.Bcm2711MboxArmToVc { grant () }

request src = kl.drivers.DNetSrv dst = kl.drivers.Bcm2711MboxArmToVc  { grant () }
response src = kl.drivers.Bcm2711MboxArmToVc dst = kl.drivers.DNetSrv { grant () }

request src = kl.drivers.USB dst = kl.drivers.Bcm2711MboxArmToVc  { grant () }
response src = kl.drivers.Bcm2711MboxArmToVc dst = kl.drivers.USB{ grant () }
```

2. Всем реализациям компонента VFS теперь необходим доступ к программе `kl.EntropyEntity`.

- Если для создания init-описания решения (файл `init.yaml`) используется шаблон `init.yaml.in` и для процессов, использующих компонент VFS (как поставляемых в составе SDK `kl.VfsNet`, `kl.VfsRamFs`, `kl.VfsSdCardFs`, так и включающих в себя VFS статически), использованы макросы `@INIT_ProgramName_ENTITY_CONNECTIONS+@` или `@INIT_ProgramName_ENTITY_CONNECTIONS@`, то изменений в init-описании не требуется.

  Иначе, если IPC-каналы для процессов, использующих компонент VFS, указаны вручную, то необходимо добавить процесс `kl.EntropyEntity` в init-описание и определить IPC-каналы между ним и процессами, использующими компонент VFS:

```
- name: kl.VfsSdCardFs
  path: VfsSdCardFs
  connections:
  ...
  - target: kl.EntropyEntity
    id: kl.EntropyEntity

- name: kl.VfsNet
  path: VfsNet
  connections:
  ...
  - target: kl.EntropyEntity
    id: kl.EntropyEntity

- name: kl.ProgramWithEmbeddedVfs
  path: ProgramWithEmbedVfs
  connections:
  ...
  - target: kl.EntropyEntity
    id: kl.EntropyEntity

- name: kl.EntropyEntity
  path: Entropy
```

- Необходимо добавить процесс `kl.EntropyEntity` в файл `security.psl` и разрешить процессам, использующим компонент VFS, и ядру взаимодействовать с ним:

```
...
use kl.EntropyEntity._
```

```
...

execute src = Einit dst = kl.drivers.EntropyEntity { grant () }
...
request src = kl.EntropyEntity dst = kl.core.Core  { grant () }
response src = kl.core.Core dst = kl.EntropyEntity { grant () }

request src = kl.VfsNet dst = kl.EntropyEntity { grant () }
response src = kl.EntropyEntity  dst = kl.VfsNet { grant () }

request src = kl.VfsSdCardFs dst = kl.EntropyEntity { grant () }
response src = kl.EntropyEntity dst = kl.VfsSdCardFs { grant () }

request src = kl.ProgramWithEmbeddedVfs dst = kl.EntropyEntity   { grant () }
response src = kl.EntropyEntity  dst = kl.ProgramWithEmbeddedVfs { grant () }
```

3. Драйверу `kl.drivers.USB` теперь необходим доступ к программе `kl.core.NameServer`.

   - Если для создания init-описания решения (файл `init.yaml`) используется шаблон `init.yaml.in` и для процесса `kl.drivers.USB` использованы макросы `@INIT_ProgramName_ENTITY_CONNECTIONS+@` или `@INIT_ProgramName_ENTITY_CONNECTIONS@`, то изменений в init-описании не требуется.

     Иначе, если IPC-каналы для процесса `kl.drivers.USB` указаны вручную, то необходимо добавить процесс `kl.core.NameServer` в init-описание и определить IPC-каналы между ним и процессом `kl.drivers.USB`:

```
- name: kl.core.NameServer
  path: ns

- name: kl.drivers.USB
  path: usb
  connections:
  ...
  - target: kl.core.NameServer
    id: kl.core.NameServer
```

   - Необходимо добавить процесс `kl.core.NameServer` в файл `security.psl` и разрешить процессу `kl.drivers.USB` и ядру взаимодействовать с ним:

```
...
use kl.core.NameServer
...

execute src = Einit dst = kl.core.NameServer { grant () }
...
request src = kl.core.NameServer dst = kl.core.Core { grant () }
response src = kl.core.Core dst = kl.core.NameServer { grant () }

request src = kl.drivers.USB dst = kl.core.NameServer { grant () }
response src = core.NameServer dst = kl.drivers.USB { grant () }
```

4. В SDK добавлена возможность использования динамических библиотек. Теперь любое решение по умолчанию собирается с использованием динамической компоновки. Это может повлиять на сборку решений, содержащих библиотеки, имеющие как статический так и динамический вариант.

- Чтобы включить принудительную статическую компоновку исполняемых файлов, нужно в корневом CMakeLists.txt проекта заменить `initialize_platform()` на `initialize_platform (FORCE_STATIC)`.

- Для перехода от статической компоновки к динамической компоновке необходимо выполнить дополнительные действия, описанные в статье "Использование динамических библиотек".

- Для использования динамических библиотек необходимо включить в решение системную программу BlobContainer.

- Необходимо добавить процесс `kl.bc.BlobContainer` в файл `security.psl` и разрешить процессам, использующим динамические библиотеки, взаимодействовать с ним:

```
...
use kl.bc.BlobContainer
...

execute src = Einit dst = kl.bc.BlobContainer  { grant () }

request
{
    /* Allows tasks with the kl.bc.BlobContainer class to send requests to
specified tasks. */
    match src = kl.bc.BlobContainer
    {
        match dst = kl.core.Core     { grant () }
        match dst = kl.VfsSdCardFs   { grant () }
    }
    /* Allows task with the kl.bc.BlobContainer class to recive request from any
task. */
    match dst = kl.bc.BlobContainer  { grant () }
}

response
{
    /* Allows tasks with the kl.bc.BlobContainer class to get responses from
specified tasks. */
    match dst = kl.bc.BlobContainer
    {
        match src = kl.core.Core     { grant () }
        match src = kl.VfsSdCardFs   { grant () }
    }
    /* Allows task with the kl.bc.BlobContainer class to send response to any task.
*/
    match src = kl.bc.BlobContainer  { grant () }
}
```

Можно вынести разрешения для работы `kl.bc.BlobContainer` в отдельный .psl файл и подключать его. (см. пример secure_logger в составе SDK).

5. Монтирование файловой системы romfs теперь возможно только в режиме readonly.

- При монтировании romfs в C/C++ коде с использованием функции `mount()` нужно передавать флаг `MS_RDONLY`.

- Также требуется внести изменения в аргументы командной строки программы VFS в init-описании или в файле CMakeLists.txt для сборки программы Einit.

Пример монтирования файловой системы romfs в файле init.yaml:

```
- name: kl.VfsSdCardFs
  path: VfsSdCardFs
  connections:
  - target: kl.drivers.SDCard
    id: kl.drivers.SDCard
  - target: kl.EntropyEntity
    id: kl.EntropyEntity

  args:
        - -l
        - nodev /tmp ramfs 0
        - -l
        - romfs /etc romfs ro
  env:
        ROOTFS: mmc0,0 / fat32 0
        VFS_FILESYSTEM_BACKEND: server:kl.VfsSdCardFs
```

Пример монтирования файловой системы romfs в файле CMakeLists.txt:

```
set (VFS_NET_ARGS "
    - -l
    - devfs /dev devfs 0
    - -l
    - romfs /etc romfs ro")

set_target_properties (${precompiled_vfsVfsNet} PROPERTIES
                       EXTRA_ARGS ${VFS_NET_ARGS})
```

Обзор KasperskyOS

KasperskyOS – специализированная операционная система на основе микроядра разделения и монитора безопасности.

См. также:

- Что нового
- О KasperskyOS Community Edition
- Системные требования
- Начало работы

Общие сведения

Микроядерность

KasperskyOS является микроядерной операционной системой. Ядро предоставляет минимальную функциональность, включая планирование исполнения программ, управление памятью и вводом-выводом. Код драйверов устройств, файловых систем, сетевых протоколов и другого системного ПО исполняется в пользовательском режиме (вне контекста ядра).

Процессы и службы

ПО, управляемое KasperskyOS, исполняется в виде процессов. *Процесс* – это запущенная на исполнение программа, которая имеет следующие особенности:

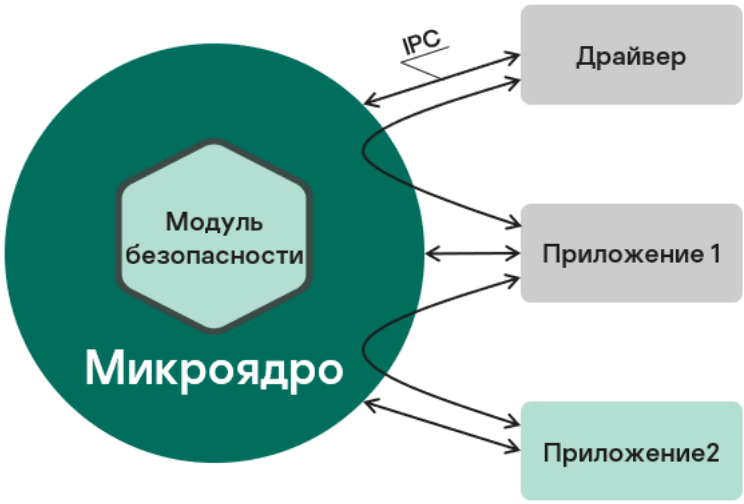
- может предоставлять службы другим процессам и/или использовать службы других процессов через механизм IPC;
- использует службы ядра через механизм IPC;
- ассоциируется с правилами безопасности, которые регулируют взаимодействия процесса с другими процессами и ядром.

Служба (англ. endpoint) – набор связанных по смыслу методов, доступных через механизм IPC (например, служба для приема и передачи данных по сети, служба для работы с прерываниями).

Реализация архитектурных подходов MILS и FLASK

Разрабатывая систему на базе KasperskyOS, ПО проектируют как набор компонентов (программ), взаимодействие между которыми регулируется механизмами безопасности. С точки зрения безопасности уровень доверия к каждому компоненту может быть высоким или низким, то есть ПО системы включает доверенные и недоверенные компоненты. Взаимодействиями компонентов между собой (и с ядром) управляет ядро (см. рис. ниже), уровень доверия к которому является высоким. Такой дизайн системы базируется на архитектурном подходе MILS (Multiple Independent Levels of Security), который применяется при разработке информационных систем ответственного применения.

Решение о разрешении или запрете конкретного взаимодействия принимает модуль безопасности Kaspersky Security Module. (Это решение называется *решением модуля безопасности*.) Модуль безопасности является модулем ядра, уровень доверия к которому является высоким, как и к самому ядру. Ядро выполняет решение модуля безопасности. Такое разделение функций по управлению взаимодействиями основано на архитектурном подходе FLASK (Flux Advanced Security Kernel), используемом в операционных системах для гибкого применения политик безопасности.



Взаимодействие процессов между собой и с ядром в KasperskyOS

Решение на базе KasperskyOS

Системное ПО (включая ядро KasperskyOS и модуль безопасности Kaspersky Security Module) и прикладное ПО, интегрированные для работы в составе программно-аппаратного комплекса, представляют собой *решение на базе KasperskyOS* (далее также *решение*). Программы, входящие в решение на базе KasperskyOS, являются *компонентами решения на базе KasperskyOS* (далее *компонентами решения*). Каждый экземпляр компонента решения исполняется в контексте отдельного процесса.

Политика безопасности решения на базе KasperskyOS

Разрешения и запреты взаимодействий процессов между собой и с ядром KasperskyOS задает *политика безопасности решения на базе KasperskyOS* (далее *политика безопасности решения*, *политика*). Политика безопасности решения сохраняется в модуле безопасности Kaspersky Security Module и используется этим модулем, когда он принимает решения о разрешении или запрете взаимодействий.

Также политика безопасности решения может задавать логику обработки обращений процесса к модулю безопасности через *интерфейс безопасности*. Процесс может использовать интерфейс безопасности, чтобы передать в модуль безопасности какие-либо данные (например, чтобы повлиять на последующие решения модуля безопасности) или получить решение модуля безопасности, которое требуется процессу для определениях своих дальнейших действий.

Технология Kaspersky Security System

Технология Kaspersky Security System позволяет реализовать разнообразные политики безопасности решений. При этом можно комбинировать несколько механизмов безопасности и гибко регулировать взаимодействия процессов между собой и с ядром KasperskyOS. На основе описания политики безопасности решения создается модуль безопасности Kaspersky Security Module для использования в конкретном решении.

Генераторы исходного кода

Часть исходного кода решения на базе KasperskyOS создается генераторами исходного кода. Специальные программы генерируют исходный код на языке C из декларативных описаний. Генерируется исходный код модуля безопасности Kaspersky Security Module, *инициализирующей программы* (запускает остальные программы в решении и статически задает топологию взаимодействия между ними), а также методов и типов для осуществления IPC (*транспортный код*).

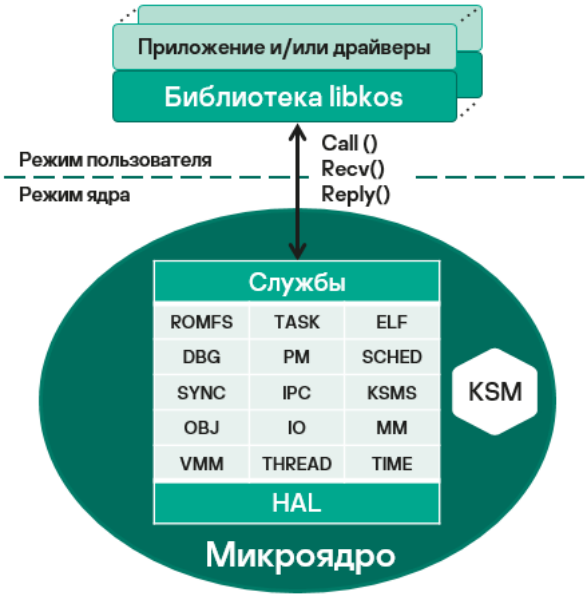
Транспортный код генерируется компилятором `nk-gen-c` на основе формальных спецификаций компонентов решения.

Исходный код модуля безопасности Kaspersky Security Module генерируется компилятором `nk-psl-gen-c` из описания политики безопасности решения и формальных спецификаций компонентов решения.

Исходный код инициализирующей программы генерируется утилитой `einit` из init-описания и формальных спецификаций компонентов решения.

Архитектура KasperskyOS

Архитектура KasperskyOS представлена на рисунке ниже:



Архитектура KasperskyOS

В KasperskyOS приложения и драйверы взаимодействуют между собой и с ядром, используя библиотеку `libkos`, которая предоставляет интерфейсы для обращения к службам ядра. (Драйвер в KasperskyOS в общем случае работает на том же уровне привилегий, что и приложение.) Библиотека `libkos` обращается к ядру, выполняя только три системных вызова: `Call()`, `Recv()` и `Reply()`, которые реализуют механизм IPC. Службы ядра поддерживаются подсистемами ядра, назначение которых приведено в таблице ниже. Подсистемы ядра взаимодействуют с аппаратурой через уровень аппаратных абстракций (англ. Hardware Abstraction Layer, HAL), что упрощает портирование KasperskyOS на различные платформы.

Подсистемы ядра и их назначение

Обозначение	Наименование	Назначение
HAL	Подсистема аппаратных абстракций	Базовая поддержка аппаратуры: таймеры, контроллеры прерываний, блок управления памятью (англ. Memory Management Unit, MMU). Подсистема включает в себя драйверы UART и низкоуровневые средства управления электропитанием.

IO	Менеджер ввода-вывода	Регистрация и освобождение ресурсов аппаратной платформы, необходимых для работы драйверов: прерываний (англ. Interrupt ReQuest, IRQ), MMIO-памяти (англ. Memory-Mapped Input-Output), портов ввода-вывода, буферов DMA. При наличии на аппаратной платформе блока управления памятью для операций ввода-вывода (англ. Input-Output Memory Management Unit, IOMMU) подсистема обеспечивает гарантию разделения памяти, используемой устройствами.
MM	Менеджер физической памяти	Выделение и освобождение физических страниц памяти, распределение областей физически непрерывных страниц.
VMM	Менеджер виртуальной памяти	Управление физической и виртуальной памятью: резервирование, фиксация, освобождение. Работа с таблицами страниц памяти для изоляции адресных пространств процессов.
THREAD	Менеджер потоков	Управление потоками исполнения: создание, завершение, блокирование и возобновление.
TIME	Подсистема часов реального времени	Получение и установка системного времени. Использование таймеров, предоставляемых аппаратурой.
SCHED	Планировщик	Планирование потоков исполнения: стандартных потоков, потоков реального времени, потоков бездействия (IDLE).
SYNC	Подсистема, обеспечивающая примитивы синхронизации	Реализация базовых примитивов синхронизации: спинлоков (англ. spinlock), мьютексов (англ. mutex), событий (англ. event). Ядро поддерживает только один примитив – фьютекс (англ. futex), остальные примитивы реализованы на его основе в пространстве пользователя.
IPC	Подсистема межпроцессного взаимодействия	Реализация синхронного механизма IPC по принципу рандеву.
KSMS	Подсистема взаимодействия с модулем безопасности	Подсистема, работающая с модулем безопасности. Она предоставляет модулю безопасности для проверки все сообщения, передающиеся через IPC.
OBJ	Менеджер объектов	Управление общим поведением всех ресурсов KasperskyOS: отслеживание жизненного цикла, назначение уникальных идентификаторов безопасности (подробнее см. "Управление доступом к ресурсам"). Подсистема тесно связана с механизмом управления доступом на основе мандатных ссылок (англ. Object Capability, OCap).
ROMFS	Подсистема запуска образа неизменяемой файловой системы	Операции с файлами из ROMFS: открытие и закрытие, получение списка файлов и их описаний, получение характеристик файла (имени, размера).
TASK	Подсистема управления процессами	Управление процессами: создание, запуск, завершение. Получение сведений о запущенных процессах (например, имени, пути) и кодов их завершения.
ELF	Подсистема загрузки исполняемых файлов	Загрузка исполняемых ELF-файлов из ROMFS в оперативную память, разбор заголовков ELF-файлов.

DBG	Подсистема поддержки отладки	Механизм отладки на основе GDB (GNU Debugger). Наличие подсистемы в ядре опционально.
PM	Менеджер электропитания	Управление электропитанием: выполнение перезагрузки и выключения.

IPC

Механизм IPC

Обмен IPC-сообщениями

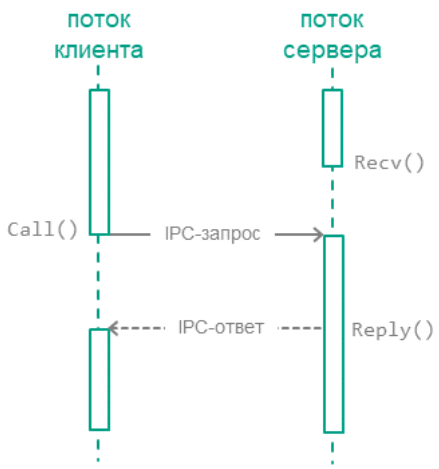
В KasperskyOS процессы взаимодействуют между собой, обмениваясь IPC-сообщениями: *IPC-запросом* и *IPC-ответом*. Во взаимодействии процессов выделяется две роли: *клиент* (процесс, инициирующий взаимодействие) и *сервер* (процесс, обрабатывающий обращение). При этом процесс, являющийся клиентом в одном взаимодействии, может выступать как сервер в другом.

Чтобы обмениваться IPC-сообщениями, клиент и сервер используют три системных вызова: `Call()`, `Recv()` и `Reply()` (см. рис. ниже):

1. Клиент направляет серверу IPC-запрос. Для этого один из потоков исполнения клиента выполняет системный вызов `Call()` и блокируется до получения IPC-ответа от сервера.

2. Серверный поток, выполнивший системный вызов `Recv()`, находится в ожидании IPC-запросов. При получении IPC-запроса этот поток разблокируется, обрабатывает запрос и отправляет IPC-ответ с помощью системного вызова `Reply()`.

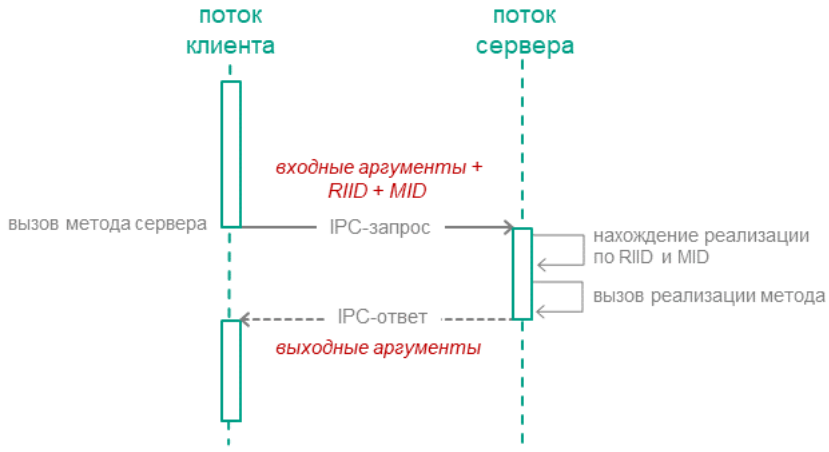
3. При получении IPC-ответа клиентский поток разблокируется и продолжает исполнение.



Обмен IPC-сообщениями между клиентом и сервером

Вызов методов служб сервера

Отправка IPC-запросов серверу осуществляется, когда клиент вызывает *методы служб* (далее также *интерфейсные методы*) сервера (см. рис. ниже). IPC-запрос содержит входные параметры вызываемого метода, а также идентификатор службы RIID и идентификатор вызываемого метода MID. Получив запрос, сервер использует эти идентификаторы, чтобы найти реализацию метода. Сервер вызывает реализацию метода, передав в нее входные параметры из IPC-запроса. Обработав запрос, сервер отправляет клиенту IPC-ответ, содержащий выходные параметры метода.



Вызов метода службы сервера

IPC-каналы

Чтобы два процесса могли обмениваться IPC-сообщениями, между ними должен быть установлен *IPC-канал*. IPC-канал имеет клиентскую и серверную стороны. Один процесс может использовать одновременно несколько IPC-каналов. При этом для одних IPC-каналов процесс может быть сервером, а для других IPC-каналов этот же процесс может быть клиентом.

В KasperskyOS предусмотрено два способа создания IPC-каналов:

1. Статический способ заключается в том, что родительский процесс создает IPC-канал между дочерними процессами. Как правило, статическое создание IPC-каналов выполняет инициализирующая программа.

2. Динамический способ позволяет уже запущенным процессам создать IPC-каналы между собой.

Управление IPC

Модуль безопасности Kaspersky Security Module интегрирован в механизм, реализующий IPC. Структура IPC-сообщений для всех возможных взаимодействий известно модулю безопасности, так как для генерации исходного кода этого модуля используются IDL-, CDL-, EDL-описания. Это позволяет модулю безопасности проверять взаимодействие процессов на соответствие политике безопасности решения.

Ядро KasperskyOS обращается к модулю безопасности каждый раз, когда один процесс отправляет IPC-сообщение другому процессу. При этом сценарий работы модуля безопасности включает следующие шаги:

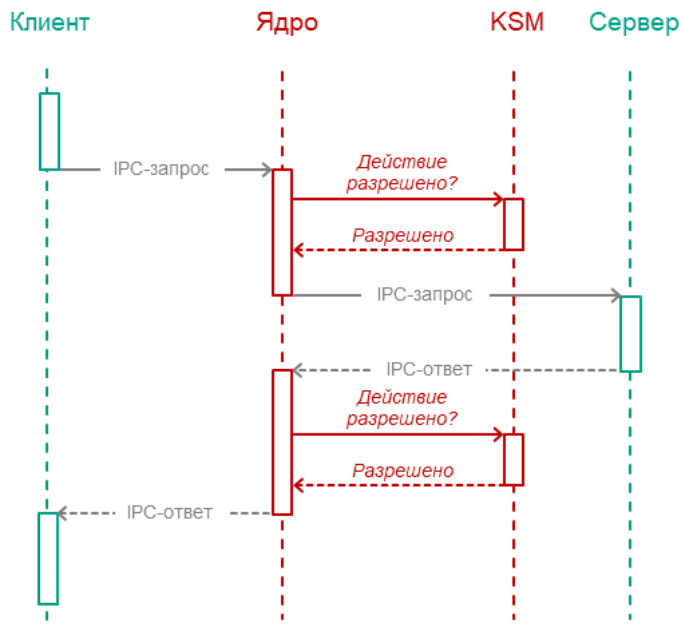
1. Модуль безопасности проверяет, что IPC-сообщение соответствует вызываемому методу службы (проверяются размер IPC-сообщения, а также размер и размещение некоторых структурных элементов).

2. Если IPC-сообщение некорректно, модуль безопасности выносит решение "запрещено", и следующий шаг сценария не выполняется. Если IPC-сообщение корректно, выполняется следующий шаг сценария.

3. Модуль безопасности проверяет, что правила безопасности разрешают запрашиваемое действие. Если это так, модуль безопасности выносит решение "разрешено", в противном случае он выносит решение "запрещено".

Ядро выполняет решение модуля безопасности, то есть доставляет IPC-сообщение процессу-получателю либо отклоняет его доставку. В случае отклонения доставки IPC-сообщения процесс-отправитель получает код ошибки через код возврата системного вызова `Call()` или `Reply()`.

Проверке подлежат как IPC-запросы, так и IPC-ответы. На рисунке ниже показана схема управляемого обмена IPC-сообщениями между клиентом и сервером.



Управляемый обмен IPC-сообщениями между клиентом и сервером

Транспортный код для IPC

Чтобы реализовать взаимодействие процессов, необходим транспортный код, отвечающий за формирование, отправку, прием и обработку IPC-сообщений. Разработчику решения на базе KasperskyOS нет необходимости самостоятельно писать транспортный код. Вместо этого можно использовать специальные инструменты и библиотеки, поставляемые в составе KasperskyOS SDK.

Транспортный код для разрабатываемых компонентов решения

Разработчик компонента решения на базе KasperskyOS может сгенерировать транспортный код на основе IDL-, CDL-, EDL-описаний, относящихся к этому компоненту. Для этого в составе KasperskyOS SDK поставляется компилятор `nk-gen-c`. Компилятор `nk-gen-c` позволяет генерировать транспортные методы и типы для использования как клиентом, так и сервером.

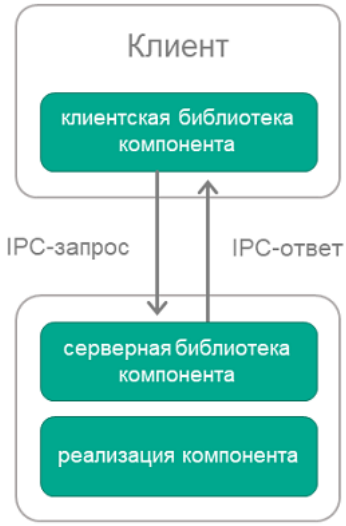
Транспортный код для поставляемых компонентов решения

Большинство компонентов, поставляемых в составе KasperskyOS SDK, может быть использовано в решении как локально, то есть путем статической компоновки с другими компонентами, так и через IPC.

Чтобы использовать поставляемый компонент через IPC, в составе KasperskyOS SDK есть следующие *транспортные библиотеки*.

- *клиентская библиотека компонента решения*, которая преобразует локальные вызовы в IPC-запросы;
- *серверная библиотека компонента решения*, которая преобразует IPC-запросы в локальные вызовы.

Клиентская библиотека компонуется с кодом клиента (с кодом компонента, который будет использовать поставляемый компонент). Серверная библиотека компонуется с реализацией поставляемого компонента (см. рис. ниже).



Использование поставляемого компонента решения через IPC

IPC между процессом и ядром

Механизм IPC используется при взаимодействии процессов с ядром KasperskyOS, то есть процессы обмениваются с ядром IPC-сообщениями. Ядро предоставляет службы, а процессы используют их. Процессы обращаются к службам ядра, вызывая функции библиотеки `libkos` (непосредственно или через другие библиотеки). Клиентский транспортный код для взаимодействия процесса с ядром сосредоточен в этой библиотеке.

Разработчику решения не требуется создавать IPC-каналы между процессами и ядром, так как эти каналы создаются автоматически при создании процессов. (Для организации взаимодействия между процессами разработчику решения нужно позаботиться о создании IPC-каналов между ними.)

Модуль безопасности Kaspersky Security Module принимает решения о взаимодействии процессов с ядром так же, как и о взаимодействии процессов между собой. (В составе KasperskyOS SDK есть IDL-, CDL-, EDL-описания для ядра, которые используются для генерации исходного кода модуля безопасности.)

Управление доступом к ресурсам

Виды ресурсов

В KasperskyOS есть два вида ресурсов:

- *Системные ресурсы*, которыми управляет ядро. К ним относятся, например, процессы, регионы памяти, прерывания.

- *Пользовательские ресурсы*, которыми управляют процессы. Примеры пользовательских ресурсов: файлы, устройства ввода-вывода, накопители данных.

Дескрипторы

Как системные, так и пользовательские ресурсы идентифицируются *дескрипторами* (англ. handles). Процессы (и ядро KasperskyOS) могут передавать дескрипторы другим процессам. Получая дескриптор, процесс получает доступ к ресурсу, который этот дескриптор идентифицирует. То есть процесс, получивший дескриптор, может запрашивать операции над ресурсом, указывая в запросе полученный дескриптор. Один и тот же ресурс может идентифицироваться несколькими дескрипторами, которые используют разные процессы.

Идентификаторы безопасности (SID)

Для системных и пользовательских ресурсов ядро KasperskyOS назначает идентификаторы безопасности. *Идентификатор безопасности* (англ. Security Identifier, SID) – это глобальный уникальный идентификатор ресурса (то есть у ресурса есть только один SID, а дескрипторов может быть несколько). Модуль безопасности Kaspersky Security Module идентифицирует ресурсы по их SID.

> При передаче IPC-сообщения, содержащего дескрипторы, ядро так изменяет это сообщение, что на этапе проверки модулем безопасности оно содержит значения SID вместо дескрипторов. Когда IPC-сообщение будет доставлено получателю, оно будет содержать дескрипторы.
> У ядра так же, как и у ресурсов, есть SID.

Контекст безопасности

Технология Kaspersky Security System позволяет применять механизмы безопасности, которые принимают на вход значения SID. При применении таких механизмов модуль безопасности Kaspersky Security Module различает ресурсы (и ядро KasperskyOS) и связывает с ними контексты безопасности. *Контекст безопасности* представляет собой данные, ассоциированные с SID, которые используются модулем безопасности для принятия решений.

Содержимое контекста безопасности зависит от используемых механизмов безопасности. Контекст безопасности может содержать, например, состояние ресурса, уровни целостности субъектов и/или объектов доступа. Если контекст безопасности хранит состояние ресурса, это позволяет, например, разрешить выполнять операции над ресурсом, если только этот ресурс находится в каком-либо конкретном состоянии.

Модуль безопасности может изменить контекст безопасности, когда принимает решение. Например, могут измениться сведения о состоянии ресурса (модуль безопасности проверил по контексту безопасности, что файл находится в состоянии "не используется", разрешил открыть файл на запись и записал в контекст безопасности этого файла новое состояние "открыт на запись").

Управление доступом к ресурсам ядром KasperskyOS

Ядро KasperskyOS управляет доступом к ресурсам одновременно двумя взаимодополняющими способами: выполняя решения модуля безопасности Kaspersky Security Module и реализуя механизм безопасности на основе мандатных ссылок (англ. Object Capability, OCap).

Каждый дескриптор ассоциируется с правами доступа к идентифицируемому им ресурсу, то есть является *мандатной ссылкой* (англ. *capability*) в терминах OCap. Получая дескриптор, процесс получает права доступа к ресурсу, который этот дескриптор идентифицирует. Например, правами доступа могут быть: право на чтение, право на запись, право на передачу другому процессу возможности выполнять операции над ресурсом (право на передачу дескриптора).

Процессы, которые используют ресурсы, предоставляемые ядром или другими процессами, являются *потребителями ресурсов*. Когда потребитель ресурсов открывает системный ресурс, ядро передает ему дескриптор, ассоциированный с правами доступа к этому ресурсу. Эти права доступа назначаются ядром. Перед выполнением операции над системным ресурсом, которую запрашивает потребитель, ядро проверяет, что у потребителя достаточно прав. Если это не так, ядро отклоняет запрос потребителя.

В IPC-сообщении дескриптор передается вместе с маской прав. *Маска прав дескриптора* представляет собой значение, биты которого интерпретируются как права доступа к ресурсу, который этот дескриптор идентифицирует. Потребитель может узнать свои права доступа к системному ресурсу из маски прав дескриптора этого ресурса. Ядро использует маску прав дескриптора для проверки, что запрашиваемые потребителем операции над системным ресурсом разрешены.

Модуль безопасности может проверять маски прав дескрипторов и по результатам проверки разрешать или запрещать взаимодействия процессов между собой и с ядром, связанные с доступом к ресурсам.

Ядро запрещает расширение прав доступа при передаче дескрипторов между процессами (при передаче дескриптора права доступа могут быть только ограничены).

Управление доступом к ресурсам поставщиками ресурсов

Процессы, которые управляют пользовательскими ресурсами и доступом к этим ресурсам для других процессов, являются *поставщиками ресурсов*. (Поставщиками ресурсов являются, например, драйверы.) Поставщики управляют доступом к ресурсам двумя взаимодополняющими способами: выполняя решения модуля безопасности Kaspersky Security Module и используя механизм OCap, который предоставляется ядром KasperskyOS.

Если обращение к ресурсу осуществляется по его имени (например, для открытия), то модуль безопасности не может быть использован для управления доступом к ресурсу без участия поставщика. Это связано с тем, что модуль безопасности идентифицирует ресурс по SID, а не по имени. В таких случаях поставщик находит у себя дескриптор ресурса по имени ресурса и передает этот дескриптор (вместе с другими данными, например, с требуемым состоянием ресурса) модулю безопасности через интерфейс безопасности (модуль безопасности получает SID, соответствующий переданному дескриптору). Модуль безопасности принимает решение и возвращает его поставщику. Поставщик выполняет решение модуля безопасности.

Когда потребитель ресурсов открывает пользовательский ресурс, поставщик передает ему дескриптор, ассоциированный с правами доступа к этому ресурсу. При этом поставщик решает, какими именно правами доступа к ресурсу будет обладать потребитель. Перед выполнением операции над пользовательским ресурсом, которую запрашивает потребитель, поставщик проверяет, что у потребителя достаточно прав. Если это не так, поставщик отклоняет запрос потребителя.

Потребитель может узнать свои права доступа к пользовательскому ресурсу из маски прав дескриптора этого ресурса. Поставщик использует маску прав дескриптора для проверки, что запрашиваемые потребителем операции над пользовательским ресурсом разрешены.

Структура и запуск решения на базе KasperskyOS

Структура решения

Загружаемый в аппаратуру образ решения на базе KasperskyOS содержит следующие файлы:

- образ ядра KasperskyOS;
- файл с исполняемым кодом модуля безопасности Kaspersky Security Module;
- исполняемый файл инициализирующей программы;
- исполняемые файлы всех остальных компонентов решения (например, прикладных программ, драйверов);
- файлы, используемые программами (например, динамические библиотеки, файлы с параметрами, шрифтами, графическими и звуковыми данными).

Для хранения файлов в образе решения используется файловая система ROMFS.

Запуск решения

Запуск решения на базе KasperskyOS происходит следующим образом:

1. Загрузчик запускает ядро KasperskyOS.
2. Ядро находит и загружает модуль безопасности (как модуль ядра).
3. Ядро запускает инициализирующую программу.
4. Инициализирующая программа запускает программы, входящие в решение (одну, несколько или все).

Начало работы

Этот раздел содержит информацию, необходимую для начала работы с KasperskyOS Community Edition.

Использование Docker-контейнера

Для установки и использования KasperskyOS Community Edition можно использовать Docker-контейнер, в котором развернут образ одной из поддерживаемых операционных систем.

Чтобы использовать Docker-контейнер для установки KasperskyOS Community Edition:

1. Убедитесь что программное обеспечение Docker установлено и запущено.
2. Для загрузки официального Docker-образа операционной системы Ubuntu GNU/Linux 22.04 "Jammy Jellyfish" из публичного репозитория Docker Hub выполните следующую команду:
   `docker pull ubuntu:22.04`
3. Для запуска образа выполните следующую команду:
   `docker run --net=host --user root --privileged -it --rm ubuntu:22.04 bash`
4. Скопируйте deb-пакет для установки KasperskyOS Community Edition в контейнер.
5. Установите KasperskyOS Community Edition.
6. Для корректной работы некоторых примеров необходимо добавить внутри контейнера директорию `/usr/sbin` в переменную окружения `PATH`, выполнив следующую команду:
   `export PATH=/usr/sbin:$PATH`

Установка и удаление

Установка

KasperskyOS Community Edition поставляется в виде deb-пакета. Для установки KasperskyOS Community Edition мы рекомендуем использовать установщик пакетов `apt`.

Для развертывания пакета с помощью `apt` запустите команду:

```
$ sudo apt update && sudo apt install <путь-к-deb-пакету>
```

Пакет будет установлен в директорию `/opt/KasperskyOS-Community-Edition-<version>`.

Чтобы удобно работать с инструментами, поставляемыми в составе KasperskyOS Community Edition SDK, нужно добавить в переменную окружения `PATH` путь к исполняемым файлам этих инструментов `/opt/KasperskyOS-Community-Edition-<version>/toolchain/bin`. Чтобы не делать это каждый раз при входе в пользовательскую сессию, нужно выполнить скрипт `/opt/KasperskyOS-Community-Edition-<version>/set_env.sh`, выйти и снова войти в сессию.

Синтаксис команды вызова скрипта `set_env.sh`:

```
$ sudo ./set_env.sh [-h] [-d]
```

Параметры:

- `-d`

  Отменяет действие скрипта.

- `-h`, `--help`

  Отображает текст справки.

Помимо изменения переменной окружения `PATH` скрипт задает переменные окружения `KOSCEVER` и `KOSCEDIR`, которые содержат версию и абсолютный путь к KasperskyOS Community Edition SDK соответственно. Использование этих переменных окружения позволяет системе сборки при запуске определить путь установки SDK, а также проверить, что версия решения соответствует версии SDK.

Удаление

Перед удалением KasperskyOS Community Edition отмените действие скрипта `set_env.sh`, если выполняли этот скрипт после установки SDK.

Для удаления KasperskyOS Community Edition выполните команду:

```
$ sudo apt remove --purge kasperskyos-community-edition
```

В результате выполнения этой команды будут удалены все установленные файлы в директории `/opt/KasperskyOS-Community-Edition-<version>`.

Настройка среды разработки

В этом разделе содержится краткое руководство по настройке среды разработки и добавлению заголовочных файлов, поставляемых в KasperskyOS Community Edition, в проект разработки.

Настройка редактора кода

Для упрощения процесса разработки решений на базе KasperskyOS перед началом работы рекомендуется:

- Установить в редакторе кода расширения и плагины для используемых языков программирования (C и/или C++).
- Добавить заголовочные файлы, поставляемые в KasperskyOS Community Edition, в проект разработки.

  Заголовочные файлы расположены в следующей директории: `/opt/KasperskyOS-Community-Edition-<version>/sysroot-aarch64-kos/include`.

Пример настройки Visual Studio Code

Например, работа с исходным кодом при разработке под KasperskyOS может проводиться в Visual Studio Code.

Для более удобной навигации по коду проекта, включая системный API, необходимо выполнить следующие действия:

1. Создайте новую рабочую область (workspace) или откройте существующую рабочую область в Visual Studio Code.

   Рабочая область может быть открыта неявно, с помощью пунктов меню `File` > `Open folder`.

2. Убедитесь, что расширение C/C++ for Visual Studio Code установлено.

3. В меню `View` выберите пункт `Command Palette`.

4. Выберите пункт `C/C++: Edit Configurations (UI)`.

5. В поле `Include path` добавьте путь `/opt/KasperskyOS-Community-Edition-<version>/sysroot-aarch64-kos/include`.

6. Закройте окно `C/C++ Configurations`.

Сборка и запуск примеров

Сборка примеров

Сборка примеров осуществляется с помощью системы сборки `CMake`, входящей в состав KasperskyOS Community Edition.

Код примеров и скрипты для сборки находятся по следующему пути:

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

> Сборку примеров нужно выполнять в директории, к которой у вас есть доступ на запись, например, в домашней.

Сборка примеров для запуска на QEMU

Чтобы выполнить сборку примера, перейдите в директорию с примером и выполните команду:

```
$ ./cross-build.sh
```

В результате работы скрипта `cross-build.sh` создается образ решения на базе KasperskyOS, который включает пример, и инициируется запуск примера на QEMU. Файл образа решения `kos-qemu-image` сохраняется в директории `<название примера>/build/einit`.

Сборка примеров для запуска на Raspberry Pi 4 B

Чтобы выполнить сборку примера, перейдите в директорию с примером и выполните команду:

```
$ ./cross-build.sh --target {kos-image|sd-image}
```

Какой образ создается в результате работы скрипта `cross-build.sh` зависит от выбора значения параметра `target`:

- `kos-image`

  Создается образ решения на базе KasperskyOS, который включает в себя пример. Файл образа решения `kos-image` сохраняется в директории `<название примера>/build/einit`.

- `sd-image`

  Создается образ файловой системы загрузочной SD-карты. В образ файловой системы загружаются: образ `kos-image`, загрузчик U-Boot, который запускает пример, и встроенное программное обеспечение (англ. firmware) для Raspberry Pi 4 B. Исходный код загрузчика U-Boot и встроенное программное обеспечение загружаются с сайта https://github.com. Файл образа файловой системы `rpi4kos.img` сохраняется в директории `<название примера>/build`.

Запуск примеров на QEMU

Запуск примеров на QEMU в Linux с графической оболочкой

Запуск примера на QEMU в Linux с графической оболочкой осуществляется скриптом `cross-build.sh`, который также выполняет сборку примера. Чтобы запустить скрипт, перейдите в директорию с примером и выполните команду:

```
$ sudo ./cross-build.sh
```

Запуск примеров на QEMU в Linux без графической оболочки

Чтобы запустить пример на QEMU в Linux без графической оболочки, перейдите в директорию с примером, соберите пример и выполните следующие команды:

```
$ cd build/einit
# Перед выполнением следующей команды убедитесь, что путь к
# директории с исполняемым файлом qemu-system-aarch64 сохранен в
# переменной окружения PATH. В случае отсутствия
# добавьте его в переменную PATH.
$ qemu-system-aarch64 -m 2048 -machine vexpress-a15,secure=on -cpu cortex-a72 -
nographic -monitor none -smp 4 -nic user -serial stdio -kernel kos-qemu-image
```

Подготовка Raspberry Pi 4 B к запуску примеров

Коммутация компьютера и Raspberry Pi 4 B

Чтобы видеть вывод с Raspberry Pi 4 B на компьютере, выполните следующие действия:

1. Соедините пины преобразователя USB-UART на базе FT232 с соответствующими GPIO-пинами Raspberry Pi 4 B (см. рис. ниже).

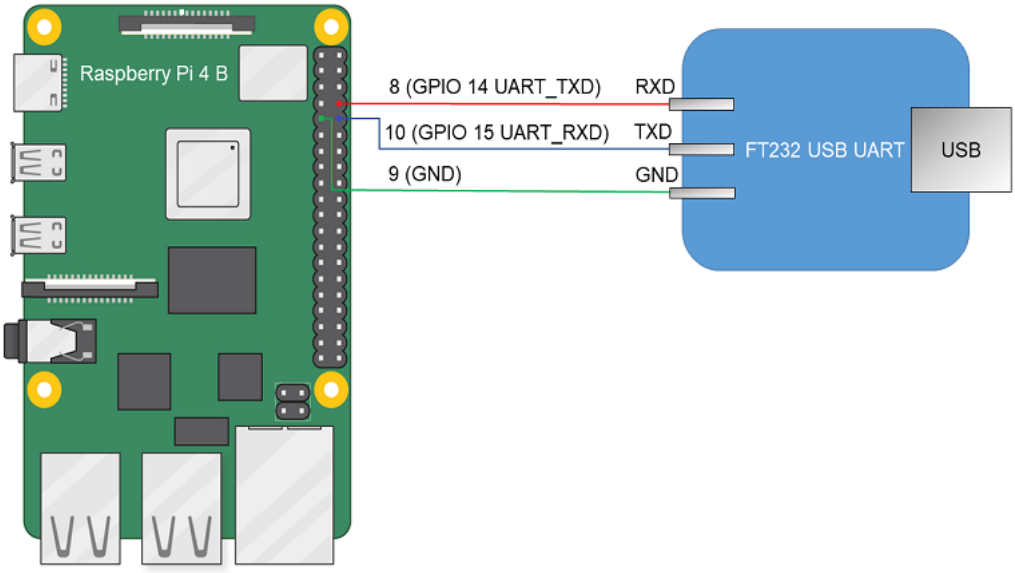


Схема соединения преобразователя USB-UART и Raspberry Pi 4 B

2. Соедините USB-порт компьютера и преобразователь USB-UART.

3. Установите PuTTY или другую аналогичную программу. Настройте параметры следующим образом: `bps = 115200`, `data bits = 8`, `stop bits = 1`, `parity = none`, `flow control = none`. Задайте порт USB, через который подключен преобразователь USB-UART, используемый для получения вывода с Raspberry Pi 4 B.

Чтобы компьютер и Raspberry Pi 4 B могли взаимодействовать через сеть Ethernet, выполните следующие действия:

1. Соедините сетевые карты компьютера и Raspberry Pi 4 B с коммутатором или друг с другом.

2. Выполните настройку сетевой карты компьютера, чтобы ее IP-адрес был в одной подсети с IP-адресом сетевой карты Raspberry Pi 4 B (параметры сетевой карты Raspberry Pi 4 B задаются в файле `dhcpcd.conf`, который находится по пути `<название примера>/resources/...`).

Подготовка загрузочной SD-карты для Raspberry Pi 4 B

Если при сборке примера был создан образ `rpi4kos.img`, то достаточно записать получившийся образ на SD-карту. Для этого подключите SD-карту к компьютеру и выполните следующую команду:

```
# В следующей команде path_to_img – путь к файлу образа,
# [X] – последний символ в имени блочного устройства для SD-карты.
$ sudo pv -L 32M path_to_img | sudo dd bs=64k of=/dev/sd[X] conv=fsync
```

Если при сборке примера был создан образ `kos-image`, то перед записью образа на SD-карту, её нужно дополнительно подготовить. Загрузочную SD-карту для Raspberry Pi 4 B можно подготовить автоматически и вручную.

Чтобы подготовить загрузочную SD-карту автоматически, подключите SD-карту к компьютеру и выполните следующие команды:

```
# Для создания файла образа загрузочного носителя (*.img)
# выполните скрипт:
$ sudo /opt/KasperskyOS-Community-Edition-
<version>/common/rpi4_prepare_sdcard_image.sh
# В следующей команде path_to_img – путь к файлу образа
# загрузочного носителя (этот путь выводится по окончании
# выполнения предыдущей команды), [X] – последний символ
# в имени блочного устройства для SD-карты.
$ sudo pv -L 32M path_to_img | sudo dd bs=64k of=/dev/sd[X] conv=fsync
```

Чтобы подготовить загрузочную SD-карту вручную, выполните следующие действия:

1. Выполните сборку загрузчика U-Boot для платформы ARMv8, который будет автоматически запускать пример. Для этого выполните следующие команды:

```
$ sudo apt install git build-essential libssl-dev bison flex unzip parted gcc-
aarch64-linux-gnu pv -y
$ git clone --depth 1 --branch v2022.01 https://github.com/u-boot/u-boot.git u-
boot-armv8
$ cd u-boot-armv8
$ make ARCH=arm CROSS_COMPILE=aarch64-linux-gnu- rpi_4_defconfig
$ echo 'CONFIG_SERIAL_PROBE_ALL=y' > ./.custom_config
$ echo 'CONFIG_BOOTCOMMAND="fatload mmc 0 ${loadaddr} kos-image; bootelf
${loadaddr} ${fdt_addr}"' >> ./.custom_config
$ echo 'CONFIG_PREBOOT="pci enum;"' >> ./.custom_config
$ ./scripts/kconfig/merge_config.sh '.config' '.custom_config'
$ make ARCH=arm CROSS_COMPILE=aarch64-linux-gnu- u-boot.bin
```

2. Подготовьте образ с файловой системой для SD-карты.

```
# Образ будет содержать boot-раздел на 1 ГБ в fat32 и три раздела по 350 МБ в ext2,
ext3 и ext4 соответственно.
$ fs_image_name=sdcard.img
$ dd if=/dev/zero of=${fs_image_name} bs=1024k count=2048
$ sudo parted ${fs_image_name} mklabel msdos
$ loop_device=$(sudo losetup --find --show --partscan ${fs_image_name})
$ sudo parted ${loop_device} mkpart primary fat32 8192s 50%
$ sudo parted ${loop_device} mkpart extended 50% 100%
$ sudo parted ${loop_device} mkpart logical ext2 50% 67%
$ sudo parted ${loop_device} mkpart logical ext3 67% 84%
$ sudo parted ${loop_device} mkpart logical ext4 84% 100%
$ sudo parted ${loop_device} set 1 boot on
$ sudo mkfs.vfat ${loop_device}p1
$ sudo mkfs.ext2 ${loop_device}p5
$ sudo mkfs.ext3 ${loop_device}p6
$ sudo mkfs.ext4 -O ^64bit,^extent ${loop_device}p7
```

3. Скопируйте загрузчик U-Boot и встроенное программное обеспечение (англ. firmware) для Raspberry Pi 4 B на полученный образ файловой системы, выполнив следующие команды:

```
# В следующих командах путь ~/mnt/fat32 используется для примера.
# Вы можете использовать другой путь.
```

```
$ mount_temp_dir=~/mnt/fat32
$ mkdir -p ${mount_temp_dir}
$ sudo mount ${loop_device}p1 ${mount_temp_dir}
$ git clone --depth 1 --branch 1.20220331
https://github.com/raspberrypi/firmware.git firmware
$ sudo cp u-boot.bin ${mount_temp_dir}/u-boot.bin
$ sudo cp -r firmware/boot/. ${mount_temp_dir}
```

4. Заполните конфигурационный файл для загрузчика U-Boot в образе используя следующие команды:

```
$ sudo sh -c "echo '[all]' > ${mount_temp_dir}/config.txt"
$ sudo sh -c "echo 'arm_64bit=1' >> ${mount_temp_dir}/config.txt"
$ sudo sh -c "echo 'enable_uart=1' >> ${mount_temp_dir}/config.txt"
$ sudo sh -c "echo 'kernel=u-boot.bin' >> ${mount_temp_dir}/config.txt"
$ sudo sh -c "echo 'dtparam=i2c_arm=on' >> ${mount_temp_dir}/config.txt"
$ sudo sh -c "echo 'dtparam=i2c=on' >> ${mount_temp_dir}/config.txt"
$ sudo sh -c "echo 'dtparam=spi=on' >> ${mount_temp_dir}/config.txt"
$ sudo sh -c "echo 'device_tree_address=0x2eff5b00' >>
${mount_temp_dir}/config.txt"
$ sudo sh -c "echo 'device_tree_end=0x2f0f5b00' >> ${mount_temp_dir}/config.txt"
$ sudo sh -c "echo 'dtoverlay=uart5' >> ${mount_temp_dir}/config.txt"
$ sudo umount ${mount_temp_dir}
$ sudo losetup -d ${loop_device}
```

5. Запишите получившийся образ на SD-карту. Для этого подключите SD-карту к компьютеру и выполните следующую команду:

```
# В следующей команде [X] – последний символ в имени блочного устройства для SD-
карты.
$ sudo pv -L 32M ${fs_image_name} | sudo dd bs=64k of=/dev/sd[X] conv=fsync
```

Запуск примеров на Raspberry Pi 4 B

Чтобы запустить пример на Raspberry Pi 4 B, выполните следующие действия:

1. Перейдите в директорию с примером и соберите пример.
2. Убедитесь, что Raspberry Pi 4 B и загрузочная SD-карта подготовлены к запуску примеров.
3. Подключите загрузочную SD-карту к Raspberry Pi 4 B.
4. Подайте питание на Raspberry Pi 4 B и дождитесь, пока запустится пример.

   О том, что пример запустился, свидетельствует вывод, отображаемый на компьютере, к которому подключен Raspberry Pi 4 B.

Запуск процессов

Обзор: Einit и init.yaml

Инициализирующая программа Einit

При запуске решения ядро KasperskyOS находит в образе решения и запускает исполняемый файл с именем `Einit`, то есть инициализирующую программу. Инициализирующая программа выполняет следующие действия:

- создает и запускает процессы при запуске решения;
- создает IPC-каналы между процессами при запуске решения (статически создает IPC-каналы).

Процесс инициализирующей программы имеет класс `Einit`.

Генерация исходного кода инициализирующей программы

В составе KasperskyOS SDK поставляется утилита `einit`, которая позволяет генерировать исходный код инициализирующей программы на языке C. Стандартным способом использования утилиты `einit` является интеграция ее вызова в один из шагов сборочного скрипта, в результате которого генерируется файл `einit.c`, содержащий исходный код инициализирующей программы. На одном из следующих шагов сборочного скрипта необходимо скомпилировать файл `einit.c` в исполняемый файл `Einit` и включить в образ решения.

> Для инициализирующей программы не требуется создавать файлы формальной спецификации. Эти файлы поставляются в составе KasperskyOS SDK и автоматически применяются при сборке решения. Однако класс процесса `Einit` должен быть указан в файле `security.psl`.

Утилита `einit` генерирует исходный код инициализирующей программы основе *init-описания*, представляющего собой текстовый файл, который обычно имеет имя `init.yaml`.

Синтаксис init.yaml

Init-описание содержит данные в формате YAML, которые идентифицируют:

- Процессы, исполнение которых начинается при запуске решения.
- IPC-каналы, которые создаются при запуске решения и используются процессами для взаимодействия между собой (не с ядром).

Эти данные представляют собой словарь с ключом `entities`, содержащий список словарей процессов. Ключи словаря процесса приведены в таблице ниже.

Ключи словаря процесса в init-описании

Ключ	Обязательный	Значение
`name`	Да	Имя класса процесса (из EDL-описания).
`task`	Нет	Имя процесса. Если его не указать, то будет взято имя класса процесса. У каждого процесса должно быть уникальное имя. Можно запустить несколько процессов одного класса, но с разными именами.
`path`	Нет	Имя исполняемого файла в ROMFS (в образе решения). Если его не указать, то будет взято имя класса процесса без префиксов и точек. Например, процессы классов `Client` и `net.Client`, для которых не указано имя исполняемого файла, будут запущены из файла `Client`. Можно запустить несколько процессов из одного исполняемого файла.
`connections`	Нет	Список словарей IPC-каналов процесса. Этот список задает статически создаваемые IPC-каналы, клиентскими IPC-дескрипторами которых будет владеть процесс. По умолчанию этот список пуст. (Помимо статически создаваемых IPC-каналов процессы могут использовать динамически создаваемые IPC-каналы.)
`args`	Нет	Список параметров запуска программы (параметры функции `main()`). Максимальный размер одного элемента списка составляет 1024 байта.
`env`	Нет	Словарь переменных окружения программы. Ключами в этом словаре являются имена переменных окружения. Максимальный размер значения переменной окружения составляет 1024 байта.

Ключи словаря IPC-канала процесса приведены в таблице ниже.

Ключи словаря IPC-канала в init-описании

Ключ	Обязательный	Значение
`id`	Да	Имя IPC-канала, которое может быть задано как конкретным значением, так и ссылкой вида `{var: <имя константы>, include: <путь к заголовочному файлу>}`.
`target`	Да	Имя процесса, который будет владеть серверным дескриптором IPC-канала.

Примеры init-описаний

Здесь приведены примеры init-описаний, демонстрирующие различные аспекты запуска процессов.

> Система сборки может автоматически создавать init-описание на основе шаблона `init.yaml.in`.

Запуск клиента и сервера и создание IPC-канала между ними

В этом примере будут запущены процесс класса `Client` и процесс класса `Server`. Имена процессов не указаны, поэтому они будут совпадать с именами классов процессов. Имена исполняемых файлов также не указаны, они также будут совпадать с именами классов процессов. Процессы будут соединены IPC-каналом с именем `server_connection`.

init.yaml

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

Запуск процессов из заданных исполняемых файлов

В этом примере будут запущены процесс класса `Client` из исполняемого файла с именем `cl`, процесс класса `ClientServer` из исполняемого файла с именем `csr` и процесс класса `MainServer` из исполняемого файла с именем `msr`. Имена процессов не указаны, поэтому они будут совпадать с именами классов процессов.

init.yaml

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

Запуск двух процессов из одного исполняемого файла

В этом примере будут запущены процесс класса `Client` из исполняемого файла с именем `Client`, а также два процесса классов `MainServer` и `BkServer` из исполняемого файла с именем `srv`. Имена процессов не указаны, поэтому они будут совпадать с именами классов процессов.

init.yaml

```yaml
entities:
- name: Client
- name: MainServer
  path: srv
- name: BkServer
  path: srv
```

Запуск двух серверов одного класса и клиента и создание IPC-каналов между клиентом и серверами

В этом примере будут запущены процесс класса `Client` (с именем `Client`) и два процесса класса `Server` с именами `UserServer` и `PrivilegedServer`. Клиент будет соединен с серверами IPC-каналами с именами `server_connection_us` и `server_connection_ps`. Имена исполняемых файлов не указаны, поэтому они будут совпадать с именами классов процессов.

init.yaml

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

Установка параметров запуска и переменных окружения программ

В этом примере будут запущены процесс класса `VfsFirst` (с именем `VfsFirst`) и процесс класса `VfsSecond` (с именем `VfsSecond`). Программа, которая будет исполняться в контексте процесса `VfsFirst`, будет запущена с параметром `-f /etc/fstab`, а также получит переменную окружения `ROOTFS` со значением `ramdisk0,0 / ext2 0` и переменную окружения `UNMAP_ROMFS` со значением `1`. Программа, которая будет исполняться в контексте процесса `VfsSecond`, будет запущена с параметром `-l devfs /dev devfs 0`. Имена исполняемых файлов не указаны, поэтому они будут совпадать с именами классов процессов.

init.yaml

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

Запуск процессов с помощью системной программы ExecutionManager

Компонент ExecutionManager предоставляет интерфейс на языке C++ для создания, запуска и остановки процессов в решениях, построенных на базе KasperskyOS.

> Интерфейс компонента ExecutionManager не подходит для использования в коде, написанном на языке C. Для управления процессами на языке C используйте интерфейс task.h библиотеки libkos.

API компонента ExecutionManager представляет собой надстройку над IPC, которая позволяет упростить процесс разработки программ. ExecutionManager является отдельной системной программой, доступ к которой осуществляется через IPC, но при этом разработчикам предоставляется клиентская библиотека, которая скрывает необходимость использования IPC-вызовов напрямую.

Программный интерфейс компонента ExecutionManager описан в статье "Компонент ExecutionManager".

Сценарий использования компонента ExecutionManager

Здесь и далее клиентом называется приложение, использующее API компонента ExecutionManager для управления другими приложениями.

Типовой сценарий использования компонента ExecutionManager включает следующие шаги:

1. Добавление программы ExecutionManager в решение. Чтобы добавить ExecutionManager в решение, необходимо:

   - Добавить следующие команды в корневой файл CMakeLists.txt:

```
find_package (execution_manager REQUIRED)
include_directories (${execution_manager_INCLUDE})
add_subdirectory (execution_manager)
```

> Для работы программы ExecutionManager необходима программа `BlobContainer`. Эта программа автоматически добавляется в решение при добавлении ExecutionManager.

   - Компонент ExecutionManager поставляется в составе SDK в виде набора статических библиотек и заголовочных файлов и собирается под конкретное решение с помощью CMake-команды `create_execution_manager_entity()` из CMake-библиотеки `execution_manager`.

     Чтобы собрать программу ExecutionManager, необходимо в корневой директории проекта создать директорию с именем `execution_manager`, а в ней создать файл `CMakeLists.txt`, в котором содержится команда `create_execution_manager_entity()`.

     CMake-команда `create_execution_manager_entity()` принимает следующие параметры:

     Обязательный параметр `ENTITY`, в котором указывается имя исполняемого файла для программы ExecutionManager.

     Опциональные параметры:

     - `DEPENDS` - дополнительные зависимости для сборки программы ExecutionManager.
     - `MAIN_CONN_NAME` - имя IPC-канала для соединения с процессом ExecutionManager. Должно совпадать со значением переменной `mainConnection` при обращении к API ExecutionManager в коде клиента.
     - `ROOT_PATH` - путь к корневой директории для служебных файлов программы ExecutionManager, по умолчанию `"/ROOT"`.
     - `VFS_CLIENT_LIB` - имя клиентской транспортной библиотеки для подключения программы ExecutionManager к программе `VFS`.

```
include (execution_manager/create_execution_manager_entity)
create_execution_manager_entity(
    ENTITY ExecMgrEntity
    MAIN_CONN_NAME ${ENTITY_NAME}
    ROOT_PATH "/root"
    VFS_CLIENT_LIB ${vfs_CLIENT_LIB})
```

   - При сборке решения (файл CMakeLists.txt для программы Einit) добавить следующие исполняемые файлы в образ решения:

- исполняемый файл программы ExecutionManager;
- исполняемый файл программы `BlobContainer`.

2. Компоновка исполняемого файла клиента с клиентской прокси-библиотекой ExecutionManager, для чего необходимо в файле `CMakeLists.txt` для сборки клиента добавить следующую команду:

```
target_link_libraries (<имя CMake-цели для сборки клиента>
${execution_manager_EXECMGR_PROXY})
```

3. Добавление разрешений для необходимых событий в описание политики безопасности решения:

   a. Чтобы программа ExecutionManager могла запускать другие процессы, политика безопасности решения должна разрешать следующие взаимодействия для класса процессов `execution_manager.ExecMgrEntity`:

   - События безопасности вида `execute` для всех классов запускаемых процессов.
   - Доступ ко всем службам программы `VFS`.
   - Доступ ко всем службам программы `BlobContainer`.
   - Доступ к службам ядра `Sync`, `Task`, `VMM`, `Thread`, `HAL`, `Handle`, `FS`, `Notice`, `CM` и `Profiler` (их описания находятся в директории `sysroot-*-kos/include/kl/core` из состава SDK).

   b. Чтобы клиент мог обращаться к программе ExecutionManager, политика безопасности решения должна разрешать следующие взаимодействия для класса клиентского процесса:

   - Доступ к соответствующим службам программы ExecutionManager (их описания находятся в директории `sysroot-*-kos/include/kl/execution_manager` из состава SDK).

4. Использование API программы ExecutionManager в коде клиента.

   Для этого необходимо использовать заголовочный файл `component/package_manager/kos_ipc/package_manager_proxy.h`. Подробнее см. "Компонент ExecutionManager".

Обзор: программа Env

Системная программа `Env` предназначена для установки параметров запуска и переменных окружения программ. Если программа `Env` включена в решение, то процессы, соединенные IPC-каналом с процессом `Env`, при своем запуске автоматически отправляют IPC-запросы этой программе и получают параметры запуска и переменные окружения.

> Использование системной программы `Env` является устаревшим способом установки параметров запуска и переменных окружения программ. Установку параметров запуска и переменных окружения программ нужно выполнять через файл `init.yaml.in` или `init.yaml`.
> Если значение параметра запуска или переменной окружения программы задано как через программу `Env`, так и через файл `init.yaml.in` или `init.yaml`, то будет применяться значение, заданное через программу `Env`.

Чтобы использовать программу `Env` в решении, необходимо:

1. Разработать код программы `Env`, используя макросы и функции из заголовочного файла `sysroot-*-kos/include/env/env.h` из состава KasperskyOS SDK.
2. Собрать исполняемый файл программы `Env`, скомпоновав его с библиотекой `env_server` из состава KasperskyOS SDK.
3. В init-описании указать, что необходимо запустить процесс `Env` и соединить с ним другие процессы (`Env` при этом является сервером). Имя IPC-канала задается макросом `ENV_SERVICE_NAME`, определенным в заголовочном файле `env.h`.
4. Включить исполняемый файл `Env` в образ решения.

Исходный код программы Env

В исходном коде программы `Env` используются следующие макросы и функции из заголовочного файла `env.h`:

- `ENV_REGISTER_ARGS(name,argarr)` – установить параметры запуска `argarr` для программы, которая будет исполняться в контексте процесса с именем `name`.
- `ENV_REGISTER_VARS(name,envarr)` – установить переменные окружения `envarr` для программы, которая будет исполняться в контексте процесса с именем `name`.
- `ENV_REGISTER_PROGRAM_ENVIRONMENT(name,argarr,envarr)` – установить параметры запуска `argarr` и переменные окружения `envarr` для программы, которая будет исполняться в контексте процесса с именем `name`.
- `envServerRun()` – инициализировать серверную часть программы `Env`, чтобы она могла отвечать на IPC-запросы.

Примеры использования программы Env

Примеры установки параметров запуска и переменных окружения программ с помощью Env

> Использование системной программы `Env` является устаревшим способом установки параметров запуска и переменных окружения программ. Установку параметров запуска и переменных окружения программ нужно выполнять через файл `init.yaml.in` или `init.yaml`.
> Если значение параметра запуска или переменной окружения программы задано как через программу `Env`, так и через файл `init.yaml.in` или `init.yaml`, то будет применяться значение, заданное через программу `Env`.

Пример установки параметров запуска программы

Исходный код программы `Env`, которая при запуске процесса с именем `NetVfs` передаст ему два параметра запуска программы: `-l devfs /dev devfs 0` и `-l romfs /etc romfs ro`:

```
env.c
```

```
#include <env/env.h>
#include <stdlib.h>

int main(int argc, char** argv)
{
    const char* NetVfsArgs[] = {
        "-l", "devfs /dev devfs 0",
        "-l", "romfs /etc romfs ro"
    };
    ENV_REGISTER_ARGS("NetVfs", NetVfsArgs);

    envServerRun();
    return EXIT_SUCCESS;
}
```

Пример установки переменных окружения программы

Исходный код программы `Env`, которая при запуске процесса с именем `Vfs3` передаст ему две переменных окружения программы: `ROOTFS=ramdisk0,0 / ext2 0` и `UNMAP_ROMFS=1`:

env.c

```
#include <env/env.h>
#include <stdlib.h>

int main(int argc, char** argv)
{
    const char* Vfs3Envs[] = {
        "ROOTFS=ramdisk0,0 / ext2 0",
        "UNMAP_ROMFS=1"
    };

    ENV_REGISTER_VARS("Vfs3", Vfs3Envs);

    envServerRun();
    return EXIT_SUCCESS;
}
```

Файловые системы и сеть

В KasperskyOS работа с файловыми системами и сетью выполняется через отдельную системную программу, реализующую виртуальную файловую систему (англ. Virtual File System, VFS).

В составе SDK компонент VFS представлен набором исполняемых файлов, библиотек, файлов формальной спецификации и заголовочных файлов. Подробнее см. раздел "Состав компонента VFS".

Основной сценарий взаимодействия с системной программой VFS происходит следующим образом:

1. Прикладная программа соединяется IPC-каналом с системной программой VFS и при сборке компонуется с клиентской библиотекой компонента VFS.

2. В прикладном коде POSIX-вызовы для работы с файловыми системами и сетью преобразуются в вызовы функций клиентской библиотеки.

> Ввод и вывод в файловые дескрипторы для стандартных потоков ввода-вывода (stdin, stdout и stderr) также преобразуется в обращения к VFS. Если прикладная программа *не скомпонована* с клиентской библиотекой компонента VFS, то вывод в stdout невозможен. В таком случае возможен только вывод в стандартный поток ошибок (stderr), который в этом случае осуществляется без использования VFS через специальные методы ядра KasperskyOS.

3. Клиентская библиотека выполняет IPC-запросы к системной программе VFS.

4. Системная программа VFS принимает IPC-запросы и вызывает соответствующие реализации файловых систем (которые, в свою очередь могут выполнять IPC-запросы к драйверам устройств) или сетевые драйверы.

5. После обработки запроса, системная программа VFS выполняет ответы на IPC-запросы прикладной программы.

Использование нескольких программ VFS

В решение можно добавить несколько копий системной программы VFS, разделив таким образом информационные потоки разных системных и прикладных программ. Также можно разделить информационные потоки в рамках одной прикладной программы. Подробнее см. "Разделение информационных потоков с помощью VFS-бэкендов".

Включение функциональности VFS в прикладную программу

Функциональность компонента VFS можно полностью включить в прикладную программу, избавляясь при этом от необходимости передавать каждый запрос через IPC. Подробнее см. "Включение функциональности VFS в программу".

При этом использование функциональности VFS по IPC позволяет разработчику решения:

- контролировать вызовы методов для работы с сетью и файловыми системами с помощью политики безопасности решения;
- соединить несколько клиентских программ с одной программой VFS;
- соединить одну клиентскую программу с двумя программами VFS для раздельной работы с сетью и файловыми системами.

Состав компонента VFS

Компонент VFS реализует виртуальную файловую систему. В составе KasperskyOS SDK компонент VFS представлен набором исполняемых файлов, библиотек, файлов формальной спецификации и заголовочных файлов, позволяющих использовать файловые системы и/или сетевой стек.

Библиотеки VFS

CMake-пакет `vfs` содержит следующие библиотеки:

- `vfs_fs` – содержит реализации файловых систем devfs, ramfs и ROMFS, а также позволяет добавить в VFS реализации других файловых систем.
- `vfs_net` – содержит реализацию файловой системы devfs и сетевого стека.
- `vfs_imp` – содержит библиотеки `vfs_fs` и `vfs_net`.
- `vfs_remote` – клиентская транспортная библиотека, которая преобразует локальные вызовы в IPC-запросы к VFS и принимает IPC-ответы.
- `vfs_server` – серверная транспортная библиотека VFS, которая принимает IPC-запросы, преобразует их в локальные вызовы и отправляет IPC-ответы.
- `vfs_local` – используется для включения функциональности VFS в программу.

Исполняемые файлы VFS

CMake-пакет `precompiled_vfs` содержит следующие исполняемые файлы:

- `VfsRamFs`;
- `VfsSdCardFs`;
- `VfsNet`.

Исполняемые файлы `VfsRamFs` и `VfsSdCardFs` включают в себя библиотеки `vfs_server`, `vfs_fs`, `vfat` и `lwext4`. Исполняемый файл `VfsNet` включает в себя библиотеки `vfs_server` и `vfs_imp`.

Каждый из этих исполняемых файлов имеет собственные значения параметров запуска и переменных окружения по умолчанию.

Файлы формальной спецификации и заголовочные файлы VFS

В директории `sysroot-*-kos/include/kl` из состава KasperskyOS SDK находятся следующие файлы VFS:

- файлы формальной спецификации `VfsRamFs.edl`, `VfsSdCardFs.edl`, `VfsNet.edl` и `VfsEntity.edl` и сгенерированные из них заголовочные файлы;
- файл формальной спецификации `Vfs.cdl` и сгенерированный из него заголовочный файл `Vfs.cdl.h`;
- файлы формальной спецификации `Vfs*.idl` и сгенерированные из них заголовочные файлы.

API библиотеки libc, поддерживаемый VFS

Функциональность VFS доступна программам через API, предоставляемый библиотекой `libc`.

Функции, реализуемые библиотеками `vfs_fs` и `vfs_net`, приведены в таблицах ниже. Символом `*` отмечены функции, которые включаются в библиотеку `vfs_fs` опционально (в зависимости от параметров сборки библиотеки).

Функции, реализуемые библиотекой vfs_fs

`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()*	

Функции, реализуемые библиотекой vfs_net

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()	

Если в VFS нет реализации вызванной функции, возвращается код ошибки `EIO`.

Создание IPC-канала до VFS

В этом примере процесс `Client` использует файловые системы и сетевой стек, а процесс `VfsFsnet` обрабатывает IPC-запросы процесса `Client`, связанные с использованием файловых систем и сетевого стека. Такой подход используется в тех случаях, когда не требуется разделение информационных потоков, связанных с файловыми системами и сетевым стеком.

Имя IPC-канала должно задаваться макросом `_VFS_CONNECTION_ID`, определенным в заголовочном файле `sysroot-*-kos/include/vfs/defs.h` из состава KasperskyOS SDK.

Init-описание примера:

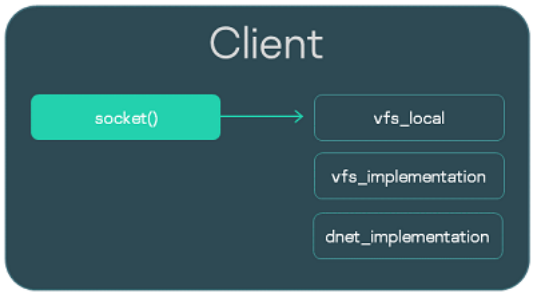
`init.yaml`

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

- name: VfsFsnet
```

Включение функциональности VFS в программу

В этом примере программа `Client` включает функциональность программы VFS для работы с сетевым стеком (см. рис. ниже).



Библиотеки компонента VFS в составе программы

Выполняется компиляция файла реализации `client.c` и компоновка с библиотеками `vfs_local`, `vfs_implementation` и `dnet_implementation`:

`CMakeLists.txt`

```
project (client)

include (platform/nk)

# Установка флагов компиляции
project_header_default ("STANDARD_GNU_11:YES" "STRICT_WARNINGS:NO")

# Генерация файла Client.edl.h
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)
```

```
# Компоновка с библиотеками VFS
target_link_libraries (Client ${vfs_LOCAL_LIB} ${vfs_IMPLEMENTATION_LIB}
${dnet_IMPLEMENTATION_LIB}
```

Если бы программа `Client` использовала файловые системы, то нужно было бы выполнить компоновку с библиотеками `vfs_local` и `vfs_fs`, а также с библиотеками реализации этих файловых систем. Кроме того, нужно было бы добавить в решение драйвер блочного устройства.

Обзор: параметры запуска и переменные окружения VFS

Параметры запуска программы VFS

- `-l <запись в формате fstab>`

  Параметр запуска `-l` монтирует заданную файловую систему.

- `-f <путь к файлу fstab>`

  Параметр `-f` монтирует файловые системы, указанные в файле `fstab`. Если переменная окружения `UNMAP_ROMFS` не определена, то поиск файла `fstab` будет выполнен в ROMFS-образе. Если переменная окружения `UNMAP_ROMFS` определена, то поиск файла `fstab` будет выполнен в файловой системе, заданной через переменную окружения `ROOTFS`.

Примеры использования параметров запуска программы VFS

Переменные окружения программы VFS

- `UNMAP_ROMFS`

  Если переменная окружения `UNMAP_ROMFS` определена, то ROMFS-образ будет удален из памяти. Это позволяет сэкономить память, а также при использовании параметра запуска `-f` дает возможность выполнить поиск файла `fstab` не в ROMFS-образе, а в файловой системе, заданной через переменную окружения `ROOTFS`.

  Пример использования переменной окружения UNMAP_ROMFS

- `ROOTFS = <запись в формате fstab>`

  Переменная окружения `ROOTFS` позволяет монтировать заданную файловую систему в корневую директорию. При использовании параметра запуска `-f` комбинация переменных окружения `ROOTFS` и `UNMAP_ROMFS` дает возможность выполнить поиск файла `fstab` не в ROMFS-образе, а в файловой системе, заданной через переменную окружения `ROOTFS`.

  Пример использования переменной окружения ROOTFS

- `VFS_CLIENT_MAX_THREADS`

  Переменная окружения `VFS_CLIENT_MAX_THREADS` позволяет переопределить параметр конфигурирования SDK `VFS_CLIENT_MAX_THREADS`.

- `_VFS_NETWORK_BACKEND=<имя VFS-бэкенда>:<имя IPC-канала до процесса VFS>`

Переменная окружения `_VFS_NETWORK_BACKEND` задает VFS-бэкенд для работы с сетевым стеком. Можно указать имя стандартного VFS-бэкенда: `client` (для программы, исполняющейся в контексте клиентского процесса), `server` (для программы VFS, исполняющейся в контексте серверного процесса) или `local`, а также имя пользовательского VFS-бэкенда. Если используется VFS-бэкенд `local`, то имя IPC-канала не указывается (`_VFS_NETWORK_BACKEND=local:`). Может быть указано более одного IPC-канала через запятую.

- `_VFS_FILESYSTEM_BACKEND=<имя VFS-бэкенда>:<имя IPC-канала до процесса VFS>`

  Переменная окружения `_VFS_FILESYSTEM_BACKEND` задает VFS-бэкенд для работы с файловыми системами. Имя VFS-бэкенда и имя IPC-канала до процесса VFS задаются так же, как и в переменной окружения `_VFS_NETWORK_BACKEND`.

Значения по умолчания для параметров запуска и переменных окружения VFS

Для исполняемого файла `VfsRamFs`:

```
ROOTFS = ramdisk0,0 / ext4 0
VFS_FILESYSTEM_BACKEND = server:kl.VfsRamFs
```

Для исполняемого файла `VfsSdCardFs`:

```
ROOTFS = mmc0,0 / fat32 0
VFS_FILESYSTEM_BACKEND = server:kl.VfsSdCardFs
-l nodev /tmp ramfs 0
-l nodev /var ramfs 0
```

Для исполняемого файла `VfsNet`:

```
VFS_NETWORK_BACKEND = server:kl.VfsNet
VFS_FILESYSTEM_BACKEND = server:kl.VfsNet
-l devfs /dev devfs 0
```

Монтирование файловых систем при запуске VFS

При запуске программы VFS по умолчанию монтируется только файловая система RAMFS в корневую директорию. Если требуется монтировать другие файловые системы, это можно сделать не только с помощью вызова функции `mount()`, но и установив параметры запуска и переменные окружения программы VFS.

> Файловые системы `ROMFS` и `squashfs` предназначены только для чтения, поэтому для монтирования этих файловых систем нужно указать параметр `ro`.

Использование параметра запуска -l

Одним из способов монтировать файловую систему является установка для программы VFS параметра запуска `-l <запись в формате fstab>`.

В этих примерах при запуске программы VFS будут монтированы файловые системы devfs и ROMFS:

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")
...
```

Использование файла fstab из ROMFS-образа

При сборке решения можно добавить файл `fstab` в ROMFS-образ. Этот файл можно использовать для монтирования файловых систем, установив для программы VFS параметр запуска `-f <путь к файлу fstab>`.

В этих примерах при запуске программы VFS будут монтированы файловые системы, заданные через файл `fstab`, который был добавлен при сборке решения в ROMFS-образ:

init.yaml.(in)

```
...
- name: VfsSecond
  args:
  - -f
  - fstab
...
```

CMakeLists.txt

```
...
set_target_properties (${vfs_ENTITY} PROPERTIES
EXTRA_ARGS
"  - -f
  - fstab")
...
```

Использование "внешнего" файла fstab

Если файл `fstab` находится не в ROMFS-образе, а в другой файловой системе, то для использования этого файла необходимо установить для программы VFS следующие параметры запуска и переменные окружения:

1. `ROOTFS`. Эта переменная окружения позволяет монтировать в корневую директорию файловую систему, содержащую файл `fstab`.

2. `UNMAP_ROMFS`. Если эта переменная окружения определена, то поиск файла `fstab` будет выполнен в файловой системе, заданной через переменную окружения `ROOTFS`.

3. `-f`. Этот параметр запуска используется, чтобы монтировать файловые системы, указанные в файле `fstab`.

В этих примерах при запуске программы VFS в корневую директорию будет монтирована файловая система ext2, в которой должен находиться файл `fstab` по пути `/etc/fstab`:

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")
...
```

Разделение информационных потоков с помощью VFS-бэкендов

В этом примере применяется паттерн безопасной разработки, предусматривающий отделение информационных потоков, связанных с использованием файловых систем, от информационных потоков, связанных с использованием сетевого стека.

Процесс `Client` использует файловые системы и сетевой стек. Процесс `VfsFirst` обеспечивает работу с файловыми системами, а процесс `VfsSecond` дает возможность работать с сетевым стеком. Через переменные окружения программ, исполняющихся в контекстах процессов `Client`, `VfsFirst` и `VfsSecond`, заданы VFS-бэкенды, которые обеспечивают раздельное использование файловых систем и сетевого стека. В результате этого IPC-запросы процесса `Client`, связанные с использованием файловых систем, обрабатываются процессом `VfsFirst`, а IPC-запросы процесса `Client`, связанные с использованием сетевого стека, обрабатываются процессом `VfsSecond` (см. рис. ниже).

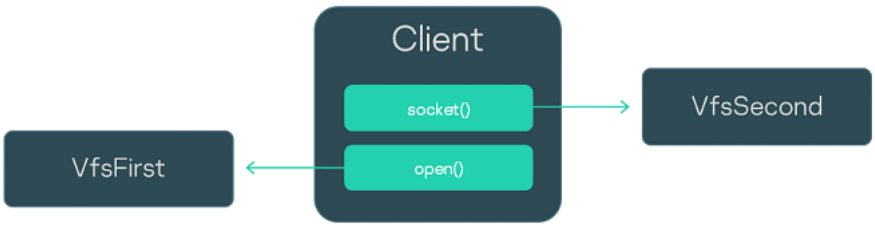


Схема взаимодействия процессов

Init-описание примера:

init.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
```

Создание VFS-бэкенда

В этом примере показано, как создать и использовать собственный VFS-бэкенд.

Процесс `Client` использует файловые системы fat32 и ext4. Процесс `VfsFirst` обеспечивает работу с файловой системой fat32, а процесс `VfsSecond` дает возможность работать с файловой системой ext4. Через переменные окружения программ, исполняющихся в контекстах процессов `Client`, `VfsFirst` и `VfsSecond`, заданы VFS-бэкенды, которые обеспечивают обработку IPC-запросов процесса `Client` процессом `VfsFirst` или `VfsSecond` в зависимости от того, какую файловую систему использует процесс `Client`. В результате этого IPC-запросы процесса `Client`, связанные с использованием файловой системы fat32, обрабатываются процессом `VfsFirst`, а IPC-запросы процесса `Client`, связанные с использованием файловой системы ext4, обрабатываются процессом `VfsSecond` (см. рис. ниже).

На стороне процесса `VfsFirst` файловая система fat32 монтируется в директорию `/mnt1`. На стороне процесса `VfsSecond` файловая система ext4 монтируется в директорию `/mnt2`. Пользовательский VFS-бэкенд `custom_client`, используемый на стороне процесса `Client`, позволяет отправлять IPC-запросы по IPC-каналу `VFS1` или `VFS2` в зависимости от того, начинается ли путь к файлу с `/mnt1` или нет. При этом пользовательский VFS-бэкенд использует в качестве посредника стандартный VFS-бэкенд `client`.

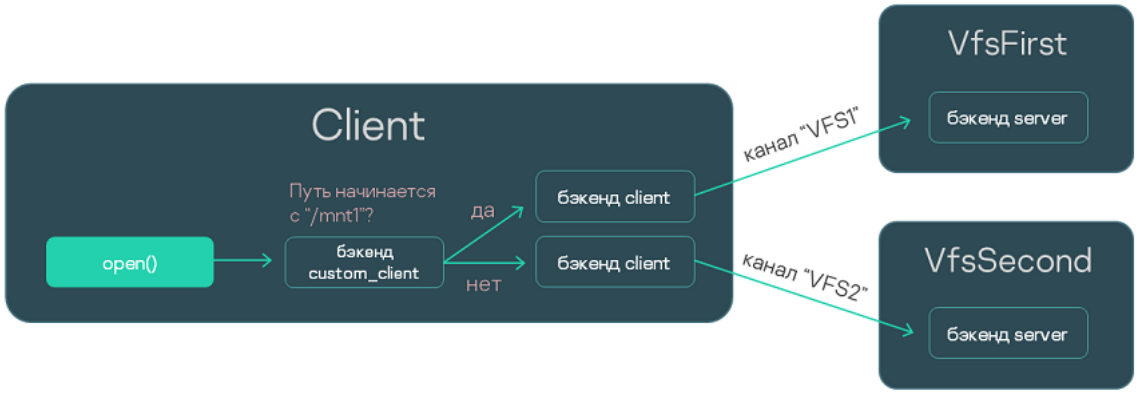


Схема взаимодействия процессов

Исходный код VFS-бэкенда

Этот файл реализации содержит исходный код VFS-бэкенда `custom_client`, использующего стандартные VFS-бэкенды `client`:

backend.c

```c
#include <vfs/vfs.h>

#include <stdio.h>
#include <stdlib.h>

#include <platform/compiler.h>
#include <pthread.h>
#include <errno.h>
#include <string.h>
#include <getopt.h>
#include <assert.h>

/* Код управления файловыми дескрипторами */
#define MAX_FDS 50

struct entry
{
    Handle handle;
    bool is_vfat;
};

struct fd_array
{
    struct entry entries[MAX_FDS];
    int pos;
    pthread_rwlock_t lock;
};

struct fd_array fds = { .pos = 0, .lock = PTHREAD_RWLOCK_INITIALIZER };

int insert_entry(Handle fd, bool is_vfat)
{
    pthread_rwlock_wrlock(&fds.lock);
    if (fds.pos == MAX_FDS)
    {
        pthread_rwlock_unlock(&fds.lock);
```

```
        return -1;
    }

    fds.entries[fds.pos].handle = fd;
    fds.entries[fds.pos].is_vfat = is_vfat;
    fds.pos++;

    pthread_rwlock_unlock(&fds.lock);
    return 0;
}

struct entry *find_entry(Handle fd)
{
    pthread_rwlock_rdlock(&fds.lock);
    for (int i = 0; i < fds.pos; i++)
    {
        if (fds.entries[i].handle == fd)
        {
            pthread_rwlock_unlock(&fds.lock);
            return &fds.entries[i];
        }
    }

    pthread_rwlock_unlock(&fds.lock);
    return NULL;
}

/* Структура пользовательского VFS-бэкенда */
struct context
{
    struct vfs wrapper;
    pthread_rwlock_t lock;

    struct vfs *vfs_vfat;
    struct vfs *vfs_ext4;
};

struct context ctx =
{
    .wrapper =
    {
        .dtor = _vfs_backend_dtor,

        .disconnect_all_clients = _disconnect_all_clients,
        .getstdin = _getstdin,
        .getstdout = _getstdout,
        .getstderr = _getstderr,

        .open = _open,
        .read = _read,
        .write = _write,
        .close = _close,
    }
};

/* Реализация методов пользовательского VFS-бэкенда */
static bool is_vfs_vfat_path(const char *path)
{
    char vfat_path[5] = "/mnt1";
    if (memcmp(vfat_path, path, sizeof(vfat_path)) != 0)
        return false;
```

```
    return true;
}

static void _vfs_backend_dtor(struct vfs *vfs)
{
    ctx.vfs_vfat->dtor(ctx.vfs_vfat);
    ctx.vfs_ext4->dtor(ctx.vfs_ext4);
}

static void _disconnect_all_clients(struct vfs *self, int *error)
{
    (void)self;
    (void)error;

    ctx.vfs_vfat->disconnect_all_clients(ctx.vfs_vfat, error);
    ctx.vfs_ext4->disconnect_all_clients(ctx.vfs_ext4, error);
}

static Handle _getstdin(struct vfs *self, int *error)
{
    (void)self;

    Handle handle = ctx.vfs_vfat->getstdin(ctx.vfs_vfat, error);
    if (handle != INVALID_HANDLE)
    {
        if (insert_entry(handle, true))
        {
            *error = ENOMEM;
            return INVALID_HANDLE;
        }
    }

    return handle;
}

static Handle _getstdout(struct vfs *self, int *error)
{
    (void)self;

    Handle handle = ctx.vfs_vfat->getstdout(ctx.vfs_vfat, error);
    if (handle != INVALID_HANDLE)
    {
        if (insert_entry(handle, true))
        {
            *error = ENOMEM;
            return INVALID_HANDLE;
        }
    }

    return handle;
}

static Handle _getstderr(struct vfs *self, int *error)
{
    (void)self;

    Handle handle = ctx.vfs_vfat->getstderr(ctx.vfs_vfat, error);
    if (handle != INVALID_HANDLE)
    {
        if (insert_entry(handle, true))
        {
            *error = ENOMEM;
```

```
            return INVALID_HANDLE;
        }
    }

    return handle;
}

static Handle _open(struct vfs *self, const char *path, int oflag, mode_t mode, int
*error)
{
    (void)self;

    Handle handle;
    bool is_vfat = false;

    if (is_vfs_vfat_path(path))
    {
        handle = ctx.vfs_vfat->open(ctx.vfs_vfat, path, oflag, mode, error);
        is_vfat = true;
    }
    else
        handle = ctx.vfs_ext4->open(ctx.vfs_ext4, path, oflag, mode, error);

    if (handle == INVALID_HANDLE)
        return INVALID_HANDLE;

    if (insert_entry(handle, is_vfat))
    {
        if (is_vfat)
            ctx.vfs_vfat->close(ctx.vfs_vfat, handle, error);
        *error = ENOMEM;
        return INVALID_HANDLE;
    }

    return handle;
}

static ssize_t _read(struct vfs *self, Handle fd, void *buf, size_t count, bool
*nodata, int *error)
{
    (void)self;

    struct entry *found_entry = find_entry(fd);

    if (found_entry != NULL && found_entry->is_vfat)
        return ctx.vfs_vfat->read(ctx.vfs_vfat, fd, buf, count, nodata, error);

    return ctx.vfs_ext4->read(ctx.vfs_ext4, fd, buf, count, nodata, error);
}

static ssize_t _write(struct vfs *self, Handle fd, const void *buf, size_t count, int
*error)
{
    (void)self;

    struct entry *found_entry = find_entry(fd);

    if (found_entry != NULL && found_entry->is_vfat)
        return ctx.vfs_vfat->write(ctx.vfs_vfat, fd, buf, count, error);

    return ctx.vfs_ext4->write(ctx.vfs_ext4, fd, buf, count, error);
}
```

```
static int _close(struct vfs *self, Handle fd, int *error)
{
    (void)self;

    struct entry *found_entry = find_entry(fd);

    if (found_entry != NULL && found_entry->is_vfat)
        return ctx.vfs_vfat->close(ctx.vfs_vfat, fd, error);

    return ctx.vfs_ext4->close(ctx.vfs_ext4, fd, error);
}

/* Конструктор пользовательского VFS-бэкенда. ctx.vfs_vfat и ctx.vfs_ext4
инициализируются
 * как стандартные бэкенды с именем "client". */
static struct vfs *_vfs_backend_create(Handle client_id, const char *config, int
*error)
{
    (void)config;

    ctx.vfs_vfat = _vfs_init("client", client_id, "VFS1", error);
    assert(ctx.vfs_vfat != NULL && "Can't initialize client backend!");
    assert(ctx.vfs_vfat->dtor != NULL && "VFS FS backend has not set the
destructor!");

    ctx.vfs_ext4 = _vfs_init("client", client_id, "VFS2", error);
    assert(ctx.vfs_ext4 != NULL && "Can't initialize client backend!");
    assert(ctx.vfs_ext4->dtor != NULL && "VFS FS backend has not set the
destructor!");

    return &ctx.wrapper;
}

/* Регистрация пользовательского VFS-бэкенда под именем custom_client */
static void _vfs_backend(create_vfs_backend_t *ctor, const char **name)
{
    *ctor = &_vfs_backend_create;
    *name = "custom_client";
}

REGISTER_VFS_BACKEND(_vfs_backend)
```

Компоновка программы Client

Создание статической библиотеки VFS-бэкенда:

CMakeLists.txt

```
...
add_library (backend_client STATIC "src/backend.c")
...
```

Компоновка программы `Client` со статической библиотеки VFS-бэкенда:

CMakeLists.txt

```
...
add_dependencies (Client vfs_backend_client backend_client)

target_link_libraries (Client
    pthread
    ${vfs_CLIENT_LIB}
    "-Wl,--whole-archive" backend_client "-Wl,--no-whole-archive" backend_client
)
...
```

Установка параметров запуска и переменных окружения программ

Init-описание примера:

init.yaml

```
entities:

- name: vfs_backend.Client
  connections:
  - target: vfs_backend.VfsFirst
    id: VFS1
  - target: vfs_backend.VfsSecond
    id: VFS2
  env:
    _VFS_FILESYSTEM_BACKEND: custom_client:VFS1,VFS2

- name: vfs_backend.VfsFirst
  args:
  - -l
  - ahci0 /mnt1 fat32 0
  env:
    _VFS_FILESYSTEM_BACKEND: server:VFS1

- name: vfs_backend.VfsSecond
  - -l
  - ahci1 /mnt2 ext4 0
  env:
    _VFS_FILESYSTEM_BACKEND: server:VFS2
```

Динамическая настройка сетевого стека

Чтобы изменить параметры сетевого стека, заданные по умолчанию, нужно использовать функцию `sysctl()` или `sysctlbyname()`, объявленные в заголовочном файле `sysroot-*-kos/include/sys/sysctl.h` из состава KasperskyOS SDK. Параметры, которые можно изменить, приведены в таблице ниже.

Настраиваемые параметры сетевого стека

Название параметра	Описание параметра
`net.inet.ip.ttl`	Максимальное время жизни (англ. Time To Live, TTL) отправляемых IP-пакетов. Не влияет на протокол ICMP.
`net.inet.ip.mtudisc`	Если имеет значение `1`, то задействован режим "Path MTU Discovery"

	(RFC 1191), влияющий на максимальный размер TCP-сегмента (англ. Maximum Segment Size, MSS). В этом режиме значение MSS определяется ограничениями узлов сети. Если режим "Path MTU Discovery" не задействован, то значение MSS не превышает заданного параметром `net.inet.tcp.mssdflt`.
`net.inet.tcp.mssdflt`	Значение MSS (в байтах), которое применяется, если только взаимодействующая сторона не сообщила это значение при открытии TCP-соединения, или не задействован режим "Path MTU Discovery" (RFC 1191). Также это значение MSS передается взаимодействующей стороне при открытии TCP-соединения.
`net.inet.tcp.minmss`	Минимальное значение MSS, в байтах.
`net.inet.tcp.mss_ifmtu`	Если имеет значение `1`, то значение MSS при открытии TCP-соединения рассчитывается, исходя из максимального размера блока передаваемых данных (англ. Maximum Transmission Unit, MTU) задействованного сетевого интерфейса. Если имеет значение `0`, то значение MSS при открытии TCP-соединения рассчитывается, исходя из MTU того сетевого интерфейса, который имеет наибольшее значение этого параметра среди всех имеющихся сетевых интерфейсов (кроме loopback-интерфейса).
`net.inet.tcp.keepcnt`	Число повторных отправок проверочных сообщений (англ. Keep-Alive Probes, KA) без ответа, после выполнения которых TCP-соединение считается закрытым. Если имеет значение `0`, то число отправок KA не ограничено.
`net.inet.tcp.keepidle`	Время неактивности TCP-соединения, по истечении которого начинают отправляться KA. Задается в условных единицах, которые можно перевести в секунды, разделив на значение параметра `net.inet.tcp.slowhz`.
`net.inet.tcp.keepintvl`	Время между повторными отправками KA при отсутствии ответа. Задается в условных единицах, которые можно перевести в секунды, разделив на значение параметра `net.inet.tcp.slowhz`.
`net.inet.tcp.recvspace`	Размер буфера для принимаемых по протоколу TCP данных, в байтах.
`net.inet.tcp.sendspace`	Размер буфера для отправляемых по протоколу TCP данных, в байтах.
`net.inet.udp.recvspace`	Размер буфера для принимаемых по протоколу UDP данных, в байтах.
`net.inet.udp.sendspace`	Размер буфера для отправляемых по протоколу UDP данных, в байтах.

Пример настройки MSS:

```
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 и транспорт

Создание IPC-каналов

Есть два способа создания IPC-каналов: статический и динамический.

Статическое создание IPC-каналов проще в реализации, поскольку для него можно использовать init-описание.

Динамическое создание IPC-каналов позволяет изменять топологию взаимодействия процессов "на лету". Это требуется, если неизвестно, какой именно сервер предоставляет службу, необходимую клиенту. Например, может быть неизвестно, на какой именно накопитель нужно будет записывать данные.

Статическое создание IPC-канала

Статическое создание IPC-каналов имеет следующие особенности:

- Клиент и сервер еще не запущены в момент создания IPC-канала.
- Создание IPC-канала выполняется родительским процессом, запускающим клиента и сервера (обычно это Einit).
- В случае удаления IPC-канал невозможно восстановить.
- Чтобы получить IPC-дескриптор и идентификатор службы (riid) после создания IPC-канала, клиент и сервер должны использовать API, определенный в заголовочном файле `sysroot-*-kos/include/coresrv/sl/sl_api.h` из состава KasperskyOS SDK.

Статически создаются IPC-каналы, заданные в init-описании.

Динамическое создание IPC-канала

Динамическое создание IPC-каналов имеет следующие особенности:

- Клиент и сервер уже запущены в момент создания IPC-канала.
- Создание IPC-канала выполняется совместно клиентом и сервером.
- Вместо удаленного может быть создан новый IPC-канал.
- Клиент и сервер получают IPC-дескриптор и идентификатор службы (riid) сразу после успешного создания IPC-канала.

Добавление в решение службы из состава KasperskyOS Community Edition

Чтобы программа `Client` могла использовать ту или иную функциональность через механизм IPC, необходимо:

1. Найти в составе KasperskyOS Community Edition исполняемый файл (условно назовем его `Server`), реализующий нужную функциональность. (Под функциональностью мы здесь понимаем одну или несколько служб, имеющих самостоятельные IPC-интерфейсы)

2. Подключить CMake-пакет, содержащий файл `Server` и его клиентскую библиотеку.

3. Добавить исполняемый файл `Server` в образ решения.

4. Изменить init-описание так, чтобы при старте решения программа `Einit` запускала новый серверный процесс из исполняемого файла `Server` и соединяла его IPC-каналом с процессом, запускаемым из файла `Client`.

   > Необходимо указать корректное имя IPC-канала, чтобы транспортные библиотеки могли идентифицировать этот канал и найти его IPC-дескрипторы. Корректное имя IPC-канала, как правило, совпадает с именем класса серверного процесса. VFS при этом является исключением.

5. Изменить PSL-описание так, чтобы разрешить запуск серверного процесса и IPC-взаимодействие между клиентом и сервером.

6. Подключить в исходном коде программы `Client` заголовочный файл с методами сервера.

7. Скомпоновать программу `Client` с клиентской библиотекой.

Пример добавления GPIO-драйвера в решение

В составе KasperskyOS Community Edition есть файл `gpio_hw`, реализующий функциональность GPIO-драйвера.

Следующие команды подключают CMake-пакет gpio:

.\CMakeLists.txt

```
...
find_package (gpio REQUIRED COMPONENTS CLIENT_LIB ENTITY)
include_directories (${gpio_INCLUDE})
...
```

Добавление исполняемого файла `gpio_hw` в образ решения производится с помощью переменной `gpio_HW_ENTITY`, имя которой можно найти в конфигурационном файле пакета – `/opt/KasperskyOS-Community-Edition-<version>/sysroot-aarch64-kos/lib/cmake/gpio/gpio-config.cmake:`

einit\CMakeLists.txt

```
...
set (ENTITIES Client ${gpio_HW_ENTITY})
...
```

В init-описание нужно добавить следующие строки:

init.yaml.in

```
...
- name: client.Client
  connections:
  - target: kl.drivers.GPIO
    id: kl.drivers.GPIO

- name: kl.drivers.GPIO
  path: gpio_hw
```

В PSL-описание нужно добавить следующие строки:

security.psl.in

```
...
execute src=Einit, dst=kl.drivers.GPIO
{
    grant()
}

request src=client.Client, dst=kl.drivers.GPIO
{
    grant()
}

response src=kl.drivers.GPIO, dst=client.Client
{
    grant()
}
...
```

В коде программы `Client` нужно подключить заголовочный файл, в котором объявлены методы GPIO-драйвера:

client.c

```
...
#include <gpio/gpio.h>
...
```

Наконец, нужно скомпоновать программу `Client` с клиентской библиотекой GPIO:

client\CMakeLists.txt

```
...
target_link_libraries (Client ${gpio_CLIENT_LIB})
...
```

> Для корректной работы GPIO-драйвера может понадобиться добавить в решение компонент BSP. Чтобы не усложнять этот пример, мы не рассматриваем здесь BSP. Подробнее см. пример gpio_output: `/opt/KasperskyOS-Community-Edition-<version>/examples/gpio_output`

Создание и использование собственных служб

Обзор: структура IPC-сообщения

В KasperskyOS все взаимодействия между процессами статически типизированы. Допустимые структуры IPC-сообщения определяются IDL-описаниями серверов.

IPC-сообщение (как запрос, так и ответ) содержит фиксированную часть и опционально арену.

Фиксированная часть IPC-сообщения

Фиксированная часть IPC-сообщения содержит RIID, MID и опционально параметры интерфейсных методов фиксированного размера.

Параметры фиксированного размера – это параметры, которые имеют IDL-типы фиксированного размера.

RIID и MID идентифицируют вызываемый интерфейс и метод:

- RIID (Runtime Implementation ID) является порядковым номером используемой службы в наборе служб сервера (начиная с нуля).
- MID (Method ID) является порядковым номером вызываемого метода в наборе методов используемой службы (начиная с нуля).

Тип фиксированной части IPC-сообщения генерируется компилятором NK на основе IDL-описания интерфейса. Для каждого метода интерфейса генерируется отдельная структура. Также генерируются типы `union` для хранения любого запроса к процессу, компоненту или интерфейсу. Подробнее см. Пример генерации транспортных методов и типов.

Арена IPC-сообщения

Арена IPC-сообщения (далее также *арена*) содержит параметры интерфейсных методов (и/или элементы этих параметров) переменного размера.

Параметры переменного размера – это параметры, которые имеют IDL-типы переменного размера.

Подробнее см. "Работа с ареной IPC-сообщений".

Максимальный размер IPC-сообщения

Максимальный размер IPC-сообщения определяется параметрами ядра KasperskyOS. На большинстве поддерживаемых KasperskyOS аппаратных платформ совокупный размер фиксированной части и арены IPC-сообщения не может превышать 4, 8 или 16 МБ.

Проверка структуры IPC-сообщения модулем безопасности

Перед тем как вызывать связанные с IPC-сообщением правила, подсистема Kaspersky Security Module проверяет отправляемое IPC-сообщение на корректность. Проверяются как запросы, так и ответы. Если IPC-сообщение имеет некорректную структуру, оно будет отклонено без вызова связанных с ним методов моделей безопасности.

Реализация IPC-взаимодействия

Чтобы упростить разработчику работу над реализацией IPC-взаимодействия, в составе KasperskyOS Community Edition поставляются:

- Компилятор NK, позволяющий сгенерировать транспортные методы и типы.
- Библиотека `libkos`, которая предоставляет API для работы с IPC-транспортом.

Реализация простейшего IPC-взаимодействия показана в примерах echo и ping (`/opt/KasperskyOS-Community-Edition-<version>/examples/`).

Получение IPC-дескриптора

Клиентский и серверный IPC-дескрипторы требуется получить, если для используемой службы нет готовых транспортных библиотек (например, вы написали собственную службу). Для самостоятельной работы с IPC-транспортом нужно предварительно инициализировать его с помощью метода `NkKosTransport_Init()`, передав в качестве второго аргумента IPC-дескриптор используемого канала.

Подробнее см. примеры echo и ping (`/opt/KasperskyOS-Community-Edition-<version>/examples/`).

> Для использования служб, которые реализованы в исполняемых файлах в составе KasperskyOS Community Edition, нет необходимости получать IPC-дескриптор. Вся работа с транспортом, включая получение IPC-дескрипторов, выполняется поставляемыми транспортными библиотеками.
> См. примеры gpio_*, net_*, net2_* и multi_vfs_* (`/opt/KasperskyOS-Community-Edition-<version>/examples/`).

Получение IPC-дескриптора при статическом создании канала

При статическом создании IPC-канала как клиент, так и сервер могут сразу после своего запуска получить свои IPC-дескрипторы с помощью методов `ServiceLocatorRegister()` и `ServiceLocatorConnect()`, указав имя созданного IPC-канала.

Например, если IPC-канал имеет имя `server_connection`, то на клиентской стороне необходимо вызвать:

```
#include <coresrv/sl/sl_api.h>
…
Handle handle = ServiceLocatorConnect("server_connection");
```

На серверной стороне необходимо вызвать:

```
#include <coresrv/sl/sl_api.h>
…
ServiceId id;
Handle handle = ServiceLocatorRegister("server_connection", NULL, 0, &id);
```

Подробнее см. примеры echo и ping (`/opt/KasperskyOS-Community-Edition-<version>/examples/`), а также заголовочный файл `/opt/KasperskyOS-Community-Edition-<version>/sysroot-aarch64-kos/include/coresrv/sl/sl_api.h`.

> Закрытие полученного IPC-дескриптора приведет к недоступности IPC-канала. Если IPC-дескриптор был закрыт, то получить его повторно и восстановить доступ к IPC-каналу невозможно.

Получение IPC-дескриптора при динамическом создании канала

Как клиент, так и сервер получают свои IPC-дескрипторы сразу при успешном динамическом создании IPC-канала.

Клиентский IPC-дескриптор является одним из выходных (`out`) аргументов метода `KnCmConnect()`. Серверный IPC-дескриптор является выходным аргументом метода `KnCmAccept()`. Подробнее см. заголовочный файл `/opt/KasperskyOS-Community-Edition-<version>/sysroot-aarch64-kos/include/coresrv/cm/cm_api.h`.

Если динамически созданный IPC-канал больше не требуется, его клиентский и серверный дескрипторы нужно закрыть. При необходимости IPC-канал может быть создан снова.

Получение идентификатора службы (riid)

Идентификатор службы (riid) требуется получить на клиентской стороне, если для используемой службы нет готовых транспортных библиотек (например, вы написали собственную службу). Для вызова методов сервера необходимо на клиентской стороне предварительно вызвать метод инициализации прокси-объекта, передав в качестве третьего параметра идентификатор службы. Например, для интерфейса `Filesystem`:

```
Filesystem_proxy_init(&proxy, &transport.base, riid);
```

Подробнее см. примеры echo и ping (`/opt/KasperskyOS-Community-Edition-<version>/examples/`).

> Для использования служб, которые реализованы в исполняемых файлах в составе KasperskyOS Community Edition, нет необходимости получать идентификатор службы. Вся работа с транспортом выполняется поставляемыми транспортными библиотеками.
> См. примеры gpio_*, net_*, net2_* и multi_vfs_* (`/opt/KasperskyOS-Community-Edition-<version>/examples/`).

Получение идентификатора службы при статическом создании канала

При статическом создании IPC-канала клиент может получить идентификатор нужной службы с помощью метода `ServiceLocatorGetRiid()`, указав дескриптор IPC-канала и квалифицированное имя службы. Например, если экземпляр компонента `OpsComp` предоставляет службу `FS`, то на клиентской стороне необходимо вызвать:

```
#include <coresrv/sl/sl_api.h>
…
nk_iid_t riid = ServiceLocatorGetRiid(handle, "OpsComp.FS");
```

Подробнее см. примеры echo и ping (`/opt/KasperskyOS-Community-Edition-<version>/examples/`), а также заголовочный файл `/opt/KasperskyOS-Community-Edition-<version>/sysroot-aarch64-kos/include/coresrv/sl/sl_api.h`.

Получение идентификатора службы при динамическом создании канала

Клиент получает идентификатор службы сразу при успешном динамическом создании IPC-канала. Клиентский IPC-дескриптор является одним из выходных (`out`) аргументов метода `KnCmConnect()`. Подробнее см. заголовочный файл `/opt/KasperskyOS-Community-Edition-<version>/sysroot-aarch64-kos/include/coresrv/cm/cm_api.h`.

Пример генерации транспортных методов и типов

При сборке решения компилятор NK на основе EDL-, CDL- и IDL-описаний генерирует набор специальных методов и типов, упрощающих формирование, отправку, прием и обработку IPC-сообщений.

В качестве примера рассмотрим класс процессов `Server`, предоставляющий службу `FS`, которая содержит единственный метод `Open()`:

`Server.edl`

```
entity Server

/* OpsComp - именованный экземпляр компонента Operations */
components {
    OpsComp: Operations
}
```

`Operations.cdl`

```
component Operations

/* FS - локальное имя службы, реализующей интерфейс Filesystem */
endpoints {
    FS: Filesystem
}
```

`Filesystem.idl`

```
package Filesystem

interface {
    Open(in string<256> name, out UInt32 h);
}
```

На основе этих описаний будут сгенерированы файлы `Server.edl.h`, `Operations.cdl.h`, и `Filesystem.idl.h` содержащие следующие методы и типы:

Методы и типы, общие для клиента и сервера

- **Абстрактные интерфейсы, содержащие указатели на реализации входящих в них методов.**

  В нашем примере будет сгенерирован один абстрактный интерфейс – `Filesystem`:

```
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;
```

- **Набор интерфейсных методов.**

  При вызове интерфейсного метода в запросе автоматически проставляются соответствующие значения RIID и MID.

  В нашем примере будет сгенерирован единственный интерфейсный метод `Filesystem_Open`:

```
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)
```

Методы и типы, используемые только на клиенте

- **Типы прокси-объектов.**

  Прокси-объект используется как аргумент интерфейсного метода. В нашем примере будет сгенерирован единственный тип прокси-объекта `Filesystem_proxy`:

```
typedef struct Filesystem_proxy {
            struct Filesystem base;
            struct nk_transport *transport;
            nk_iid_t iid;
        } Filesystem_proxy;
```

- **Функции для инициализации прокси-объектов.**

  В нашем примере будет сгенерирована единственная инициализирующая функция `Filesystem_proxy_init`:

```
void Filesystem_proxy_init(struct Filesystem_proxy *self,
                           struct nk_transport *transport,
```

```
                                nk_iid_t iid)
```

- **Типы, определяющие структуру фиксированной части сообщения для каждого конкретного метода.**

  В нашем примере будет сгенерировано два таких типа: `Filesystem_Open_req` (для запроса) и `Filesystem_Open_res` (для ответа).

```
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;
```

Методы и типы, используемые только на сервере

- **Тип, содержащий все службы компонента, а также инициализирующая функция. (Для каждого компонента сервера.)**

  > При наличии вложенных компонентов этот тип также содержит их экземпляры, а инициализирующая функция принимает соответствующие им инициализированные структуры. Таким образом, при наличии вложенных компонентов, их инициализацию необходимо начинать с самого вложенного.

  В нашем примере будет сгенерирована структура `Operations_component` и функция `Operations_component_init`:

```
typedef struct Operations_component {
     struct Filesystem *FS;
};

void Operations_component_init(struct Operations_component *self,
                               struct Filesystem *FS)
```

- **Тип, содержащий все службы, предоставляемые сервером непосредственно; все экземпляры компонентов, входящие в сервер; а также инициализирующая функция.**

  В нашем примере будет сгенерирована структура `Server_entity` и функция `Server_entity_init`:

```
#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)
```

- **Типы, определяющие структуру фиксированной части сообщения для любого метода конкретного интерфейса.**

  В нашем примере будет сгенерировано два таких типа: `Filesystem_req` (для запроса) и `Filesystem_res` (для ответа).

```
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;
};
```

- **Типы, определяющие структуру фиксированной части сообщения для любого метода любой службы конкретного компонента.**

  > При наличии вложенных компонентов эти типы также содержат структуры фиксированной части сообщения для любых методов любых служб, включенных во все вложенные компоненты.

  В нашем примере будет сгенерировано два таких типа: `Operations_component_req` (для запроса) и `Operations_component_res` (для ответа).

```
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;
```

- **Типы, определяющие структуру фиксированной части сообщения для любого метода любой службы конкретного компонента, экземпляр которого входит в сервер.**

  > При наличии вложенных компонентов эти типы также содержат структуры фиксированной части сообщения для любых методов любых служб, включенных во все вложенные компоненты.

  В нашем примере будет сгенерировано два таких типа: `Server_entity_req` (для запроса) и `Server_entity_res` (для ответа).

```
#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-методы (диспетчеры) для отдельного интерфейса, компонента или класса процессов.**

  Диспетчеры анализируют полученный запрос (значения RIID и MID), вызывают реализацию соответствующего метода, после чего сохраняют ответ в буфер. В нашем примере будут сгенерированы диспетчеры `Filesystem_interface_dispatch`, `Operations_component_dispatch` и `Server_entity_dispatch`.

  Диспетчер класса процессов обрабатывает запрос и вызывает методы, реализуемые этим классом. Если запрос содержит некорректный RIID (например, относящийся к другой службе, которой нет у этого класса процессов) или некорректный MID, диспетчер возвращает `NK_EOK` или `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)
```

В специальных случаях можно использовать диспетчеры интерфейса и компонента. Они принимают дополнительный аргумент – ID реализации интерфейса (`nk_iid_t`). Запрос будет обработан только если переданный аргумент и RIID из запроса совпадают, а MID корректен. В противном случае диспетчеры возвращают `NK_EOK` или `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)
```

Работа с ареной IPC-сообщений

Общие сведения об арене

С точки зрения разработчика решения на базе KasperskyOS арена IPC-сообщений представляет собой байтовый буфер в памяти процесса, предназначенный для хранения передаваемых через IPC данных переменного размера, то есть входных, выходных и error-параметров интерфейсных методов (и/или элементов этих параметров), которые имеют IDL-типы переменного размера. Также арена используется при обращении к модулю безопасности Kaspersky Security Module для хранения входных параметров методов интерфейса безопасности (и/или элементов этих параметров), которые имеют IDL-типы переменного размера. (Параметры интерфейсных методов постоянного размера хранятся в фиксированной части IPC-сообщения.) Арены используются как на стороне клиента, так и на стороне сервера. Одна арена предназначена либо для передачи, либо для приема через IPC данных переменного размера, но не для передачи и приема этих данных одновременно, то есть условно арены можно разделить на арены IPC-запросов (содержат входные параметры интерфейсных методов) и арены IPC-ответов (содержат выходные и error-параметры интерфейсных методов).

Через IPC передается только использованная часть арены, то есть занятая данными. (При отсутствии данных арена не передается.) Использованная часть арены включает один или несколько участков. В одном участке арены хранится массив объектов одного типа, например, массив однобайтовых объектов или массив структур. В разных участках арены могут храниться массивы объектов разного типа. Адрес начала арены должен быть выравнен на границу *2^N*-байтовой последовательности, где *2^N* – значение, которое больше либо равно размеру наибольшего примитивного типа в арене (например, наибольшего поля примитивного типа в структуре). Адрес участка арены также должен быть выравнен на границу *2^N*-байтовой последовательности, где *2^N* – значение, которое больше либо равно размеру наибольшего примитивного типа в участке арены.

Необходимость наличия нескольких участков в арене возникает, если интерфейсный метод имеет несколько входных, выходных или error-параметров переменного размера, а также если несколько элементов входных, выходных или error-параметров интерфейсного метода имеют переменный размер. Например, если интерфейсный метод имеет входной параметр IDL-типа `sequence` и входной параметр IDL-типа `bytes`, то в арене IPC-запросов будет как минимум два участка, но могут потребоваться и дополнительные участки, если параметр IDL-типа `sequence` состоит из элементов IDL-типа переменного размера (например, `string`, то есть элементами последовательности являются строковые буферы). Также, к примеру, если интерфейсный метод имеет один выходной параметр IDL-типа `struct`, который содержит два поля типа `bytes` и `string`, то в арене IPC-ответов будет два участка.

Из-за выравнивания адресов участков арены между этими участками могут появляться неиспользованные промежутки, поэтому размер использованной части арены может превышать размер помещенных в нее данных.

API для работы с ареной

Набор функций и макросов для работы с ареной определен в заголовочном файле `sysroot-*-kos/include/nk/arena.h` из состава KasperskyOS SDK. Также функция для копирования строки в арену объявлена в заголовочном файле `sysroot-*-kos/include/coresrv/nk/transport-kos.h` из состава KasperskyOS SDK.

Сведения о функциях и макросах, определенных в заголовочном файле `sysroot-*-kos/include/nk/arena.h`, приведены в таблице ниже. В этих функциях и макросах арена и участок арены идентифицируются дескриптором арены (тип `nk_arena`) и дескриптором участка арены (тип `nk_ptr_t`) соответственно. *Дескриптор арены* представляет собой структуру, содержащую три указателя: на начало арены, на начало неиспользованной части арены и на конец арены. *Дескриптор участка арены* представляет собой структуру, содержащую смещение участка арены в байтах (относительно начала арены) и размер участка арены в байтах. (Тип дескриптора участка арены определен в заголовочном файле `sysroot-*-kos/include/nk/types.h`. из состава KasperskyOS SDK.)

Создание арены

Чтобы передавать через IPC параметры интерфейсных методов переменного размера, нужно создать арены как на стороне клиента, так и на стороне сервера. (При обработке IPC-запросов на стороне сервера с использованием функции `NkKosDoDispatch()`, определенной в заголовочном файле `sysroot-*-kos/include/coresrv/nk/transport-kos-dispatch.h` из состава KasperskyOS SDK, арены IPC-запросов и IPC-ответов создаются автоматически.)

Чтобы создать арену, нужно создать буфер (в стеке или куче) и инициализировать дескриптор арены.

Адрес буфера должен быть выравнен так, чтобы удовлетворять максимальному размеру примитивного типа, который может быть помещен в этот буфер. Адрес буфера, созданного динамически, обычно имеет достаточное выравнивание для помещения в этот буфер данных примитивного типа максимального размера. Чтобы обеспечить требуемое выравнивание адреса статически созданного буфера, можно использовать спецификатор `alignas`.

Чтобы инициализировать дескриптор арены, используя указатель на уже созданный буфер, нужно использовать функцию или макрос API:

- макрос `NK_ARENA_INITIALIZER()`;
- функцию `nk_arena_init()`;
- функцию `nk_arena_create()`;
- макрос `NK_ARENA_FINAL()`;
- макрос `nk_arena_init_final()`.

Тип указателя не имеет значения, поскольку в коде функций и макросов API этот указатель приводится к указателю на однобайтовый объект.

Макрос `NK_ARENA_INITIALIZER()` и функции `nk_arena_init()` и `nk_arena_create()` инициализируют дескриптор арены, которая может содержать один и более участков. Макросы `NK_ARENA_FINAL()` и `nk_arena_init_final()` инициализируют дескриптор арены, которая на протяжении всего своего жизненного цикла содержит только один участок, занимающий всю арену.

Чтобы создать буфер в стеке и инициализировать дескриптор одним шагом, нужно использовать макрос `NK_ARENA_AUTO()`. Этот макрос создает арену, которая может содержать один и более участков, а адрес буфера, созданного этим макросом, имеет достаточное выравнивание для помещения в этот буфер данных примитивного типа максимального размера.

Размер арены должен быть достаточен, чтобы с учетом выравнивания адресов участков вместить параметры переменного размера для IPC-запросов или IPC-ответов одного интерфейсного метода или множества интерфейсных методов, соответствующих одному интерфейсу, компоненту или классу процессов. Автоматически генерируемый транспортный код (заголовочные файлы `*.idl.h`, `*.cdl.h`, `*.edl.h`) содержит константы `*_arena_size`, значения которых гарантированно соответствуют достаточным размерам арен в байтах.

Заголовочные файлы `*.idl.h`, `*.cdl.h`, `*.edl.h` содержат следующие константы `*_arena_size`:

- `<имя интерфейса>_<имя интерфейсного метода>_req_arena_size` – размер арены IPC-запросов для указанного интерфейсного метода указанного интерфейса;
- `<имя интерфейса>_<имя интерфейсного метода>_res_arena_size` – размер арены IPC-ответов для указанного интерфейсного метода указанного интерфейса;
- `<имя интерфейса>_req_arena_size` – размер арены IPC-запросов для любого интерфейсного метода указанного интерфейса;
- `<имя интерфейса>_res_arena_size` – размер арены IPC-ответов для любого интерфейсного метода указанного интерфейса.

Заголовочные файлы `*.cdl.h`, `*.edl.h` дополнительно содержат следующие константы `*_arena_size`:

- `<имя компонента>_component_req_arena_size` – размер арены IPC-запросов для любого интерфейсного метода указанного компонента;
- `<имя компонента>_component_res_arena_size` – размер арены IPC-ответов для любого интерфейсного метода указанного компонента.

Заголовочные файлы `*.edl.h` дополнительно содержат следующие константы `*_arena_size`:

- `<имя класса процессов>_entity_req_arena_size` – размер арены IPC-запросов для любого интерфейсного метода указанного класса процессов;
- `<имя класса процессов>_entity_res_arena_size` – размер арены IPC-ответов для любого интерфейсного метода указанного класса процессов.

Константы, содержащие размер арены IPC-запросов или IPC-ответов для одного интерфейсного метода (`<имя интерфейса>_<имя интерфейсного метода>_req_arena_size` и `<имя интерфейса>_<имя интерфейсного метода>_res_arena_size`), предназначены для использования на стороне клиента. Остальные константы могут использоваться как на стороне клиента, так и на стороне сервера.

Примеры создания арены:

```
/* Пример 1 */
alignas(8) char reqBuffer[Write_WriteInLog_req_arena_size];
struct nk_arena reqArena = NK_ARENA_INITIALIZER(
                           reqBuffer, reqBuffer + sizeof(reqBuffer));

/* Пример 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));

/* Пример 3 */
char req_buffer[kl_CliApplication_Run_req_arena_size];
```

```
struct nk_arena req_arena = nk_arena_create(req_buffer, sizeof(req_buffer));

/* Пример 4 */
nk_ptr_t ptr;
const char *cstr = "example";
nk_arena arena = NK_ARENA_FINAL(&ptr, cstr, strlen(cstr));

/* Пример 5 */
const char *path = "path_to_file";
size_t len = strlen(path);
/* Структура для сохранения фиксированной части IPC-запроса */
struct kl_VfsFilesystem_Rmdir_req req;
struct nk_arena req_arena;
nk_arena_init_final(&req_arena, &req.path, path, len);

/* Пример 6 */
struct nk_arena res_arena = NK_ARENA_AUTO(kl_Klog_component_res_arena_size);
```

Заполнение арены данными перед передачей через IPC

Перед передачей IPC-запроса на стороне клиента или IPC-ответа на стороне сервера арену нужно заполнить данными. Если для создания арены используется макрос `NK_ARENA_FINAL()` или `nk_arena_init_final()`, то резервировать участок арены не требуется, а нужно только заполнить этот участок данными. Если для создания арены используется макрос `NK_ARENA_INITIALIZER()` или `NK_ARENA_AUTO()` либо функция `nk_arena_init()` или `nk_arena_create()`, то в арене необходимо зарезервировать один или несколько участков, чтобы поместить в них данные. Чтобы зарезервировать участок арены, нужно использовать функцию или макрос API:

- функцию `__nk_arena_alloc()`;
- макрос `nk_arena_store()`;
- функцию `__nk_arena_store()`;
- макрос `nk_arena_alloc()`;
- функцию `NkKosCopyStringToArena()`.

Дескриптор участка арены, который передается через выходной параметр этих функций и макросов, а также макросов `NK_ARENA_FINAL()` и `nk_arena_init_final()`, нужно поместить в фиксированную часть либо в арену IPC-сообщения. Если интерфейсный метод имеет параметр переменного размера, то фиксированная часть IPC-сообщений вместо самого параметра содержит дескриптор участка арены, в котором находится этот параметр. Если интерфейсный метод имеет параметр постоянного размера с элементами переменного размера, то фиксированная часть IPC-сообщений вместо самих элементов параметра содержит дескрипторы участков арены, в которых находятся эти элементы параметра. Если интерфейсный метод имеет параметр переменного размера, содержащий элементы переменного размера, то фиксированная часть IPC-сообщений содержит дескриптор участка арены, в котором находятся дескрипторы других участков арены, содержащих эти элементы параметра.

Макрос `nk_arena_store()` и функции `__nk_arena_store()` и `NkKosCopyStringToArena()` не только резервируют участок арены, но и копируют данные в этот участок.

Макрос `nk_arena_alloc()` позволяет получить адрес зарезервированного участка арены. Также адрес участка арены можно получить, используя функцию `__nk_arena_get()` или макрос `nk_arena_get()`, которые дополнительно через выходной параметр передают размер арены.

Зарезервированный участок арены можно уменьшить. Для этого нужно использовать макрос `nk_arena_shrink()` или функцию `_nk_arena_shrink()`.

Чтобы отменить текущее резервирование участков арены для последующего резервирования новых участков под другие данные (после отправки IPC-сообщения), нужно вызвать функцию `nk_arena_reset()`. Если для создания арены используется макрос `NK_ARENA_FINAL()` или `nk_arena_init_final()`, то отменять резервирование участка не требуется, так как такая арена на протяжении всего своего жизненного цикла содержит один участок, занимающий всю арену.

Примеры заполнения арены данными:

```
/* Пример 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));
/* Структура для сохранения фиксированной части IPC-запроса */
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;

/* Пример 2 */
char req_arena_buf[StringMaxSize];
struct nk_arena req_arena = NK_ARENA_INITIALIZER(req_arena_buf,
                                        req_arena_buf + sizeof(req_arena_buf));
/* Структура для сохранения фиксированной части IPC-запроса */
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);

/* Пример 3 */
char reqArenaBuf[kl_core_DCM_req_arena_size];
struct nk_arena reqArena
    = NK_ARENA_INITIALIZER(reqArenaBuf,
                           reqArenaBuf + sizeof(reqArenaBuf));
/* Структура для сохранения фиксированной части IPC-запроса */
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;

/* Пример 4 */
unsigned counter = 0;
nk_ptr_t *paths;
/* Резервирование участка арены для дескрипторов других участков арены */
paths = nk_arena_alloc(nk_ptr_t, resArena, &res->logRes, msgCount);
while(...)
{
    ...
    /* Резервирование участков арены с сохранением их дескрипторов в
     * ранее зарезервированном участке арены с адресом paths */
    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++;
}
```

Получение данных из арены после приема через IPC

Перед получением IPC-запроса на стороне сервера или IPC-ответа на стороне клиента для арены, в которую будут помещены полученные через IPC-данные, нужно отменить текущее резервирование участков, вызвав функцию `nk_arena_reset()`. Это требуется сделать, даже если для создания арены используется макрос `NK_ARENA_FINAL()` или `nk_arena_init_final()`. (Макросы `NK_ARENA_INITIALIZER()` и `NK_ARENA_AUTO()`, а также функции `nk_arena_init()` и `nk_arena_create()` создают арену без зарезервированных участков. Перед однократном использовании такой арены для сохранения полученных через IPC данных вызывать функцию `nk_arena_reset()` не требуется.)

Чтобы получить указатели на участки арены и размеры этих участков, нужно использовать функцию `__nk_arena_get()` или макрос `nk_arena_get()`, передавая через входной параметр соответствующие дескрипторы участков арены, полученные из фиксированной части и арены IPC-сообщения.

Пример получения данных из арены:

```
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));
/* Структура для сохранения IPC-запроса */
struct kl_rump_DhcpcdConfig_Version_req req;
req.buflen = buflen;
/* Структура для сохранения IPC-ответа */
struct kl_rump_DhcpcdConfig_Version_res res;
/* Вызов интерфейсного метода */
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);
```

Дополнительные возможности API

Чтобы получить размер арены, нужно вызвать функцию `nk_arena_capacity()`.

Чтобы получить размер использованной части арены, нужно вызвать функцию `nk_arena_allocated_size()`.

Чтобы проверить, является ли корректным дескриптор участка арены, нужно использовать макрос `nk_arena_validate()` или функцию `__nk_arena_validate()`.

Сведения о функциях и макросах API

Функции и макросы arena.h

Функция/Макрос	Сведения о функции/макросе
`NK_ARENA_INITIALIZER()`	Назначение Инициализирует дескриптор арены. Параметры • [in] `_start` – указатель на начало арены. • [in] `_end` – указатель на конец арены. Значения макроса Код инициализации дескриптора арены.
`nk_arena_init()`	Назначение Инициализирует дескриптор арены. Параметры • [out] `self` – указатель на дескриптор арены. • [in] `start` – указатель на начало арены. • [in] `end` – указатель на конец арены. Возвращаемые значения Нет.
`nk_arena_create()`	Назначение Создает и инициализирует дескриптор арены. Параметры • [in] `start` – указатель на начало арены. • [in] `size` – размер арены в байтах. Возвращаемые значения Дескриптор арены.
`NK_ARENA_AUTO()`	Назначение Создает в стеке буфер, а также создает и инициализирует дескриптор арены. Параметры • [in] `size` – размер арены в байтах. Должен быть задан константой.

	Значения макроса Дескриптор арены.
`NK_ARENA_FINAL()`	Назначение Инициализирует дескриптор арены, содержащей только один участок. Параметры • [out] `ptr` – указатель на дескриптор участка арены. • [in] `start` – указатель на начало арены. • [in] `count` – число объектов в участке арены. Значения макроса Дескриптор арены.
`nk_arena_reset()`	Назначение Отменяет резервирование участков арены. Параметры • [in,out] `self` – указатель на дескриптор арены. Возвращаемые значения Нет.
`__nk_arena_alloc()`	Назначение Резервирует участок арены заданного размера с заданным выравниванием. Параметры • [in,out] `self` – указатель на дескриптор арены. • [out] `ptr` – указатель на дескриптор участка арены. • [in] `size` – размер участка арены в байтах. • [in] `align` – значение, задающее выравнивание участка арены. Адрес участка арены может быть невыравненным (*align=1*) или выравненным (*align=2,4,...,2^N*) на границу *2^N*-байтовой последовательности (например, двухбайтовой, четырехбайтовой). Возвращаемые значения В случае успеха возвращает `NK_EOK`, иначе возвращает код ошибки.
`nk_arena_capacity()`	Назначение Позволяет получить размер арены.

	Параметры • [in] `self` – указатель на дескриптор арены. Возвращаемые значения Размер арены в байтах. Дополнительные сведения Если параметр имеет значение `NK_NULL`, возвращает `0`.
`nk_arena_validate()`	Назначение Проверяет, является ли корректным дескриптор участка арены. Параметры • [in] `type` – тип объектов, для которых предназначен участок арены. • [in] `arena` – указатель на дескриптор арены. • [in] `ptr` – указатель на дескриптор участка арены. Значения макроса Имеет значение `0` при ненулевом размере арены, если выполняются все следующие условия: 1. Смещение, указанное в дескрипторе участка арены, не превышает размер арены. 2. Размер, указанный в дескрипторе участка арены, не превышает размер арены, уменьшенный на смещение, указанное в дескрипторе участка арены. 3. Размер, указанный в дескрипторе участка арены, кратен размеру типа объектов, для которых предназначен этот участок арены. Имеет значение `0` при нулевом размере арены, если выполняются все следующие условия: 1. Смещение, указанное в дескрипторе участка арены, равно нулю. 2. Размер, указанный в дескрипторе участка арены, равен нулю. Имеет значение `-1` при нарушении хотя бы одного условия как в случае с ненулевым, так и в случае с нулевым размером арены, или если параметр `ptr` имеет значение `NK_NULL`.
`__nk_arena_validate()`	Назначение Проверяет, является ли корректным дескриптор участка арены. Параметры

	• [in] `self` – указатель на дескриптор арены. • [in] `ptr` – указатель на дескриптор участка арены. Возвращаемые значения Возвращает `0` при ненулевом размере арены, если выполняются все следующие условия: 1. Смещение, указанное в дескрипторе участка арены, не превышает размер арены. 2. Размер, указанный в дескрипторе участка арены, не превышает размер арены, уменьшенный на смещение, указанное в дескрипторе участка арены. Возвращает `0` при нулевом размере арены, если выполняются все следующие условия: 1. Смещение, указанное в дескрипторе участка арены, равно нулю. 2. Размер, указанный в дескрипторе участка арены, равен нулю. Возвращает `-1` при нарушении хотя бы одного условия как в случае с ненулевым, так и в случае с нулевым размером арены, или если параметр `ptr` имеет значение `NK_NULL`.
`__nk_arena_get()`	Назначение Позволяет получить указатель на участок арены и размер этого участка. Параметры • [in] `self` – указатель на дескриптор арены. • [in] `ptr` – указатель на дескриптор участка арены. • [out] `size` – размер участка арены в байтах. Возвращаемые значения Указатель на участок арены или `NK_NULL`, если хотя бы один параметр имеет значение `NK_NULL`.
`nk_arena_allocated_size()`	Назначение Позволяет получить размер использованной части арены. Параметры • [in] `self` – указатель на дескриптор арены. Возвращаемые значения Размер использованной части арены в байтах.

	Дополнительные сведения Если параметр имеет значение `NK_NULL`, возвращает `0`.
`nk_arena_store()`	Назначение Резервирует участок арены для заданного числа объектов заданного типа и копирует эти объекты в зарезервированный участок. Параметры • [in] `type` – тип объектов, которые нужно скопировать в участок арены. • [in,out] `arena` – указатель на дескриптор арены. • [out] `ptr` – указатель не дескриптор участка арены. • [in] `src` – указатель на буфер с объектами, которые нужно скопировать в участок арены. • [in] `count` – число объектов, которые нужно скопировать в участок арены. Значения макроса В случае успеха имеет значение `0`, иначе имеет значение `-1`.
`__nk_arena_store()`	Назначение Резервирует участок арены с заданным выравниванием для данных заданного размера и копирует эти данные в зарезервированный участок. Параметры • [in,out] `self` – указатель на дескриптор арены. • [out] `ptr` – указатель не дескриптор участка арены. • [in] `src` – указатель на буфер с данными, которые нужно скопировать в участок арены. • [in] `size` – размер данных, которые нужно скопировать в участок арены, в байтах. • [in] `align` – значение, задающее выравнивание участка арены. Адрес участка арены может быть невыравненным (*align=1*) или выравненным (*align=2,4,...,2^N*) на границу *2^N*-байтовой последовательности (например, двухбайтовой, четырехбайтовой). Возвращаемые значения В случае успеха возвращает `0`, иначе возвращает `-1`.
`nk_arena_init_final()`	Назначение

	Инициализирует дескриптор арены, содержащей только один участок. Параметры • [out] `arena` – указатель на дескриптор арены. • [out] `ptr` – указатель на дескриптор участка арены. • [in] `start` – указатель на начало арены. • [in] `count` – число объектов, для которых предназначен участок арены. Значения макроса Нет.
`nk_arena_alloc()`	Назначение Резервирует участок арены для заданного числа объектов заданного типа. Параметры • [in] `type` – тип объектов, для которых предназначен участок арены. • [in,out] `arena` – указатель на дескриптор арены. • [out] `ptr` – указатель на дескриптор участка арены. • [in] `count` – число объектов, для которых предназначен участок арены. Значения макроса В случае успеха имеет значение адреса зарезервированного участка арены, иначе имеет значение `NK_NULL`.
`nk_arena_get()`	Назначение Позволяет получить адрес участка арены и число объектов заданного типа, которое вмещается в этом участке. Параметры • [in] `type` – тип объектов, для которых предназначен участок арены. • [in] `arena` – указатель на дескриптор арены. • [in] `ptr` – указатель на дескриптор участка арены. • [out] `count` – указатель на число объектов, которое вмещается в участке арены.

	Значения макроса В случае успеха имеет значение адреса участка арены, иначе имеет значение `NK_NULL`. Дополнительные сведения Если размер участка арены не кратен размеру типа объектов, для которых этот участок предназначен, имеет значение `NK_NULL`.
`nk_arena_shrink()`	Назначение Уменьшает размер участка арены. Параметры • [in] `type` – тип объектов, для которых предназначен уменьшенный участок арены. • [in,out] `arena` – указатель на дескриптор арены. • [in,out] `ptr` – указатель на дескриптор участка арены. • [in] `count` – число объектов, для которых предназначен уменьшенный участок арены. Значения макроса В случае успеха имеет значение адреса уменьшенного участка арены, иначе имеет значение `NK_NULL`. Дополнительные сведения Если требуемый размер участка арены превышает текущий, имеет значение `NK_NULL`. Если выравнивание участка арены, который нужно уменьшить, не удовлетворяет типу объектов, для которых предназначен уменьшенный участок, имеет значение `NK_NULL`. Если участок арены, который нужно уменьшить, является последним в арене, то освободившаяся часть этого участка становится доступной для резервирования последующих участков.
`_nk_arena_shrink()`	Назначение Уменьшает размер участка арены. Параметры • [in,out] `self` – указатель на дескриптор арены. • [in,out] `ptr` – указатель на дескриптор участка арены. • [in] `size` – размер уменьшенного участка арены в байтах.

- [in] `align` – значение, используемое функцией для проверки выравнивания участка арены, который нужно уменьшить. Адрес участка арены может быть невыравненным (*align=1*) или выравненным (*align=2,4,...,2^N*) на границу *2^N*-байтовой последовательности (например, двухбайтовой, четырехбайтовой).

Возвращаемые значения

В случае успеха возвращает адрес уменьшенного участка арены, иначе возвращает `NK_NULL`.

Дополнительные сведения

Если требуемый размер участка арены превышает текущий, возвращает `NK_NULL`.

Если выравнивание участка арены, который нужно уменьшить, не удовлетворяет заданному выравниванию, возвращает `NK_NULL`.

Если участок арены, который нужно уменьшить, является последним в арене, то освободившаяся часть этого участка становится доступной для резервирования последующих участков.

Транспортный код на языке C++

> Перед чтением этого раздела, нужно ознакомиться со сведениями о механизме IPC в KasperskyOS и об IDL-, CDL-, EDL-описаниях.

Чтобы реализовать взаимодействие процессов, необходим транспортный код, отвечающий за формирование, отправку, прием и обработку IPC-сообщений.

У разработчика решения на базе KasperskyOS нет необходимости самостоятельно писать транспортный код. Вместо этого можно использовать специальные инструменты и библиотеки, поставляемые в составе KasperskyOS SDK. Эти библиотеки позволяют разработчику компонента решения сгенерировать транспортный код на основе IDL-, CDL-, EDL-описаний, относящихся к этому компоненту.

Транспортный код

Для генерации транспортного кода на языке C++ в составе KasperskyOS SDK поставляется компилятор `nkppmeta`.

Компилятор `nkppmeta` позволяет генерировать транспортные C++ прокси-объекты (proxy) и стабы (stub) для использования как клиентом, так и сервером.

Прокси-объекты используются клиентом для упаковки параметров вызываемого метода в IPC-запрос, выполнения IPC-запроса и распаковки IPC-ответа.

Стабы используются сервером для распаковки параметров из IPC-запроса, диспетчеризации вызова на соответствующую реализацию метода и упаковки IPC-ответа.

Генерация транспортного кода для разработки на C++

Для генерации транспортных прокси-объектов и стабов с помощью генератора `nkppmeta` при сборке решения используются `CMake`-команды add_nk_idl(), add_nk_cdl() и add_nk_edl().

Типы C++ в файле *.idl.cpp.h

Каждый интерфейс определяется в IDL-описании. Это описание задает имя интерфейса, сигнатуры интерфейсных методов и типы данных для параметров интерфейсных методов.

Для генерации транспортного кода при сборке решения используются `CMake`-команда add_nk_idl(), которая создает `CMake`-цель для генерации заголовочных файлов для заданного IDL-файла при помощи компилятора `nkppmeta`.

Генерируемые заголовочные файлы содержат представление на языке C++ для интерфейса и типов данных, описанных в IDL-файле, а также методы, необходимые для использования прокси-объектов и стабов.

Соответствие типов данных, объявленных в IDL-файле, типам C++ приведены в таблице ниже.

Соответствие типов IDL типам C++

Тип IDL	Тип C++
`SInt8`	`int8_t`
`SInt16`	`int16_t`
`SInt32`	`int32_t`
`SInt64`	`int64_t`
`UInt8`	`uint8_t`
`UInt16`	`uint16_t`
`UInt32`	`uint32_t`
`UInt64`	`uint64_t`
`Handle`	`Handle` (определен в `coresrv/handle/handletype.h`)
`string`	`std::string`
`union`	`std::variant`
`struct`	`struct`
`array`	`std::array`
`sequence`	`std::vector`
`bytes`	`std::vector<std::byte>`

Работа с транспортным кодом на C++

Сценарии разработки клиента и сервера, которые обмениваются IPC-сообщениями, представлены в разделах "Статическое создание IPC-каналов при разработке на языке C++" и "Динамическое создание IPC-каналов при разработке на языке C++"

Статическое создание IPC-каналов при разработке на языке C++

Чтобы реализовать клиентскую программу, вызывающую метод службы, предоставляемой серверной программой, необходимо:

1. Подключить сгенерированный заголовочный файл описания (`*.edl.cpp.h`) клиентской программы.
2. Подключить сгенерированные заголовочные файлы описаний используемых интерфейсов (`*.idl.cpp.h`).
3. Подключить заголовочные файлы:
   - `/opt/KasperskyOS-Community-Edition-<version>/sysroot-*-kos/include/kosipc/application.h`
   - `/opt/KasperskyOS-Community-Edition-<version>/sysroot-*-kos/include/kosipc/api.h`
   - `/opt/KasperskyOS-Community-Edition-<version>/sysroot-*-kos/include/kosipc/connect_static_channel.h`
4. Инициализировать объект приложения, вызвав функцию `kosipc::MakeApplicationAutodetect()`. (Также можно использовать функции `kosipc::MakeApplication()` и `kosipc::MakeApplicationPureClient()`.)
5. Получить клиентский IPC-дескриптор канала и идентификатор службы (`riid`) вызвав функцию `kosipc::ConnectStaticChannel()`.

   Функция принимает имя IPC-канала (из файла init.yaml) и квалифицированное имя службы (из CDL- и EDL-описаний компонента решения).
6. Инициализировать прокси-объект для используемой службы, вызвав функцию `MakeProxy()`.

Пример

```
// Создание и инициализация объекта приложения
kosipc::Application app = kosipc::MakeApplicationAutodetect();

// Создание и инициализация прокси-объекта
auto proxy = app.MakeProxy<IDLInterface>(
     kosipc::ConnectStaticChannel(channelName, endpointName))

// Вызов метода требуемой службы
proxy->DoSomeWork();
```

Чтобы реализовать серверную программу, предоставляющую службы другим программам, необходимо:

1. Подключить сгенерированный заголовочный файл `*.edl.cpp.h`, содержащий описание компонентной структуры программы, включая все предоставляемые службы.
2. Подключить заголовочные файлы:
   - `/opt/KasperskyOS-Community-Edition-<version>/sysroot-*-kos/include/kosipc/event_loop.h`
   - `/opt/KasperskyOS-Community-Edition-<version>/sysroot-*-kos/include/kosipc/api.h`
   - `/opt/KasperskyOS-Community-Edition-<version>/sysroot-*-kos/include/kosipc/serve_static_channel.h`

3. Создать классы, содержащие реализации интерфейсов, которые данная программа и её компоненты предоставляют в виде служб.

4. Инициализировать объект приложения, вызвав функцию `kosipc::MakeApplicationAutodetect()`.

5. Инициализировать структуру `kosipc::components::Root`, которая описывает компонентную структуру программы и описания интерфейсов всех предоставляемых программой служб.

6. Связать поля структуры `kosipc::components::Root` с объектами, реализующими соответствующие службы.

   Поля структуры `Root` повторяют иерархию компонентов и служб, заданную совокупностью CDL- и EDL-файлов.

7. Получить серверный IPC-дескриптор канала, вызвав функцию `ServeStaticChannel()`.

   Функция принимает имя IPC-канала (из файла init.yaml) и структуру, созданную на шаге 5.

8. Создать объект `kosipc::EventLoop`, вызвав функцию `MakeEventLoop()`.

9. Запустить цикл диспетчеризации входящих IPC-сообщений, вызвав метод `Run()` объекта `kosipc::EventLoop`.

Пример

```
// Создание объектов классов, которые реализуют интерфейсы,
// предоставляемые сервером в виде служб
MyIDLInterfaceImp_1 impl_1;
MyIDLInterfaceImp_2 impl_2;

// Создание и инициализация объекта приложения
kosipc::Application app = kosipc::MakeApplicationAutodetect();

// Создание и инициализация объекта root, описывающего
// компоненты и службы сервера
kosipc::components::Root    root;

// Связывание объекта root с объектами классов, реализующими службы сервера
root.component1.endpoint1 = &impl_1;
root.component2.endpoint2 = &impl_2;

// Создание и инициализация объекта, который реализует
// цикл диспетчеризации входящих IPC-сообщений
kosipc::EventLoop    loop = app.MakeEventLoop(ServeStaticChannel(channelName, root));

// Запуск цикла в текущем потоке
loop.Run();
```

Динамическое создание IPC-каналов при разработке на языке C++

Динамическое создание IPC-канала на стороне клиента включает следующие шаги:

1. Подключить к клиентской программе сгенерированный заголовочный файл описания (`*.edl.cpp.h`).

2. Подключить сгенерированные заголовочные файлы описаний используемых интерфейсов (`*.idl.cpp.h`).

3. Подключить заголовочные файлы:

   - `/opt/KasperskyOS-Community-Edition-<version>/sysroot-*-kos/include/kosipc/application.h`
   - `/opt/KasperskyOS-Community-Edition-<version>/sysroot-*-kos/include/kosipc/make_application.h`
   - `/opt/KasperskyOS-Community-Edition-<version>/sysroot-*-kos/include/kosipc/connect_dynamic_channel.h`

4. Получить указатели на имя сервера и квалифицированное имя службы с помощью сервера имен – специального сервиса ядра, представленного программой `NameServer`. Для этого необходимо подключиться к серверу имен вызовом функции `NsCreate()` и найти сервер, предоставляющий требуемую службу, используя функцию `NsEnumServices()`. Подробнее см. "Динамическое создание IPC-каналов (cm_api.h, ns_api.h)".

5. Создать объект приложения, вызвав функцию `kosipc::MakeApplicationAutodetect()`. (Также можно использовать функции `kosipc::MakeApplication()` и `kosipc::MakeApplicationPureClient()`.)

6. Создать прокси-объект для требуемой службы, вызвав функцию `MakeProxy()`. В качестве входного параметра функции `MakeProxy()` использовать вызов функции `kosipc::ConnectDynamicChannel()`. В функцию `kosipc::ConnectDynamicChannel()` передать указатели на имя сервера и квалифицированное имя службы, полученные на шаге 4.

После успешной инициализации прокси-объекта клиенту доступен вызов методов требуемой службы.

Пример

```
NsHandle ns;

// Подключение к серверу имен
Retcode rc = NsCreate(RTL_NULL, INFINITE_TIMEOUT, &ns);

char serverName[kl_core_Types_UCoreStringSize];
char endpointName[kl_core_Types_UCoreStringSize];

// Получение указателей на имя сервера и квалифицированное имя службы
rc = NsEnumServices(
     ns, interfaceName, 0,
     serverName, kl_core_Types_UCoreStringSize,
     endpointName, kl_core_Types_UCoreStringSize);

// Создание и инициализация объекта приложения
kosipc::Application app = kosipc::MakeApplicationAutodetect();

// Создание и инициализация прокси-объекта
auto proxy = app.MakeProxy<IDLInterface>(
     kosipc::ConnectDynamicChannel(serverName, endpointName))

// Вызов метода требуемой службы
proxy->DoSomeWork();
```

Динамическое создание IPC-канала на стороне сервера включает следующие шаги:

1. Подключить к серверной программе сгенерированный заголовочный файл (`*.edl.cpp.h`), содержащий описание компонентной структуры сервера, включая все предоставляемые службы.

2. Подключить заголовочные файлы:

   - `/opt/KasperskyOS-Community-Edition-<version>/sysroot-*-kos/include/kosipc/application.h`
   - `/opt/KasperskyOS-Community-Edition-<version>/sysroot-*-kos/include/kosipc/event_loop.h`
   - `/opt/KasperskyOS-Community-Edition-<version>/sysroot-*-kos/include/kosipc/make_application.h`
   - `/opt/KasperskyOS-Community-Edition-<version>/sysroot-*-kos/include/kosipc/root_component.h`
   - `/opt/KasperskyOS-Community-Edition-<version>/sysroot-*-kos/include/kosipc/serve_dynamic_channel.h`
   - `/opt/KasperskyOS-Community-Edition-<version>/sysroot-*-kos/include/kosipc/simple_connection_acceptor.h`

3. Создать классы, содержащие реализации интерфейсов, которые сервер предоставляет в виде служб. Создать и инициализировать объекты этих классов.

4. Создать объект приложения, вызвав функцию `kosipc::MakeApplicationAutodetect()`.

5. Создать и инициализировать объект класса `kosipc::components::Root`, который описывает структуру компонентов и служб сервера. Эта структура генерируется из описаний в CDL- и EDL-файлах.

6. Связать объект класса `kosipc::components::Root` с объектами классов, созданными на шаге 3.

7. Создать и инициализировать объект класса `kosipc::EventLoop`, который реализует цикл диспетчеризации входящих IPC-сообщений, вызвав функцию `MakeEventLoop()`. В качестве входного параметра функции `MakeEventLoop()` использовать вызов функции `ServeDynamicChannel()`. В функцию `ServeDynamicChannel()` передать объект класса `kosipc::components::Root`, созданный на шаге 5.

8. Запустить цикл диспетчеризации входящих IPC-сообщений в отдельном потоке, вызвав метод `Run()` объекта `kosipc::EventLoop`.

9. Создать и инициализировать объект, который реализует обработчик приема входящих запросов на динамическое создание IPC-канала.

   При создании объекта можно использовать класс `kosipc::SimpleConnectionAcceptor`, который является стандартной реализацией интерфейса `kosipc::IConnectionAcceptor`. (Интерфейс `kosipc::IConnectionAcceptor` определен в файле `/opt/KasperskyOS-Community-Edition-<version>/sysroot-*-kos/include/kosipc/connection_acceptor.h`.) В этом случае обработчик будет реализовать следующую логику: если запрашиваемая клиентом служба опубликована на сервере, то запрос от клиента будет принят, иначе отклонен.

   Если необходимо создать собственный обработчик, то следует реализовать свою логику обработки запросов в методе `OnConnectionRequest()`, унаследованном от интерфейса `kosipc::IConnectionAcceptor`. Этот метод будет вызываться сервером при получении от клиента запроса на динамическое создание IPC-канала.

10. Создать объект класса `kosipc::EventLoop`, который реализует цикл приема входящих запросов на динамическое создание IPC-канала, вызвав функцию `MakeEventLoop()`. В качестве входного параметра функции `MakeEventLoop()` использовать вызов функции `ServeConnectionRequests()`. В функцию `ServeConnectionRequests()` передать объект, созданный на шаге 9.

Цикл приема входящих запросов на динамическое создание IPC-канала может быть только один. Цикл должен работать в одном потоке. Цикл приема входящих запросов на динамическое создание IPC-канала должен быть создан после создания цикла диспетчеризации входящих IPC-сообщений (см. на шаг 7).

11. Запустить цикл приема входящих запросов на динамическое соединение в текущем потоке, вызвав метод `Run()` объекта `kosipc::EventLoop`.

Пример

```
// Создание объектов классов, которые реализуют интерфейсы,
// предоставляемые сервером в виде служб
MyIDLInterfaceImp_1 impl_1;
MyIDLInterfaceImp_2 impl_2;

// Создание и инициализация объекта приложения
kosipc::Application app = kosipc::MakeApplicationAutodetect();

// Создание и инициализация объекта root, описывающего
// компоненты и службы сервера
kosipc::components::Root    root;

// Связывание объекта root с объектами классов, реализующими службы сервера.
// Поля объекта root повторяют описание компонентов и служб,
// заданную совокупностью CDL- и EDL-файлов.
root.component1.endpoint1 = &impl_1;
root.component2.endpoint2 = &impl_2;

// Создание и инициализация объекта, который реализует
// цикл диспетчеризации входящих IPC-сообщений
kosipc::EventLoop   loopDynamicChannel = app.MakeEventLoop(ServeDynamicChannel(root));

// Запуск цикла диспетчеризации входящих IPC-сообщений в отдельном потоке
std::thread     dynChannelThread(
    [&loopDynamicChannel]() {
        loopDynamicChannel.Run();
    }
);

// Создание объекта, реализующего стандартный обработчик приема входящих запросов
// на динамическое создание IPC-канала
kosipc::SimpleConnectionAcceptor    acceptor(root);

// Создание объекта, реализующего цикл приема входящих запросов
// на динамическое создание IPC-канала
kosipc::EventLoop   loopDynamicChannel =
app.MakeEventLoop(ServeConnectionRequests(&acceptor));

// Запуск цикла приема входящих запросов на динамическое создание IPC-канала в текущем
потоке
loopConnectionReq.Run();
```

При необходимости можно создать и инициализировать несколько объектов класса `kosipc::components::Root`, объединенных в список объектов типа `ServiceList` с помощью метода `AddServices()`. Использование нескольких объектов позволяет, например, разделять компоненты и службы сервера на группы или публиковать службы под разными именами.

Пример

```
// Создание и инициализация объекта group_1
kosipc::components::Root    group_1;

group_1.component1.endpoint1 = &impl_1;
group_1.component2.endpoint2 = &impl_2;

// Создание и инициализация объекта group_2
kosipc::components::Root    group_2;

group_2.component1.endpoint1 = &impl_3;
group_2.component2.endpoint2 = &impl_4;

// Создание и инициализация объекта group_3
kosipc::components::Root group_3;

group_3.component1.endoint1 = &impl_5;

// Создание списка объектов
ServiceList endpoints;
endpoints.AddServices(group_1);
endpoints.AddServices(group_2);
endpoints.AddServices(group_3.component1.endpoint1, "SomeCustomEndpointName");

// Создание объекта, реализующего обработчик приема входящих запросов
// на динамическое создание IPC-канала
kosipc::SimpleConnectionAcceptor    acceptor(std::move(endpoints));

// Создание объекта, реализующего цикл приема входящих запросов
// на динамическое создание IPC-канала
kosipc::EventLoop   loopDynamicChannel =
app.MakeEventLoop(ServeConnectionRequests(&acceptor));
```

Коды возврата

Общие сведения

В решении на базе KasperskyOS коды возврата функций различных API (например, API библиотек `libkos` и `kdf`, драйверов, транспортного кода, прикладного ПО) имеют тип 32-битного знакового целого числа. Этот тип определен в заголовочном файле `sysroot-*-kos/include/rtl/retcode.h` из состава KasperskyOS SDK так:

```
typedef __INT32_TYPE__ Retcode;
```

Множество кодов возврата состоит из кода успеха со значением `0` и кодов ошибок. Код ошибки интерпретируется как структура данных, формат которой описан в заголовочном файле `sysroot-*-kos/include/rtl/retcode.h` из состава KasperskyOS SDK. Этот формат предусматривает наличие нескольких полей, которые содержат не только сведения о результатах вызова функции, но и следующую дополнительную информацию:

- Флаг в поле `Customer`, сигнализирующий о том, что код ошибки определен разработчиками решения на базе KasperskyOS, а не разработчиками ПО из состава KasperskyOS SDK.

  Благодаря флагу в поле `Customer` разработчики решения на базе KasperskyOS и разработчики ПО из состава KasperskyOS SDK могут определять коды ошибок из непересекающихся множеств.

- Глобальный идентификатор кода ошибки в поле `Space`.

  Глобальные идентификаторы позволяют определять непересекающиеся множества кодов ошибок. Коды ошибок могут быть общими и специфичными. Общие коды ошибок могут использоваться в API любых компонентов решения и в API любых составных частей компонентов решения (например, драйвер или VFS могут быть составной частью компонента решения). Специфичные коды ошибок используются в API одного или нескольких компонентов решения или в API одной или нескольких составных частей компонентов решения.

  Например, идентификатору `RC_SPACE_GENERAL` соответствуют коды общих ошибок, идентификатору `RC_SPACE_KERNEL` соответствуют коды ошибок ядра, идентификатору `RC_SPACE_DRIVERS` соответствуют коды ошибок драйверов.

- Локальный идентификатор кода ошибки в поле `Facility`.

  Локальные идентификаторы позволяют определять непересекающиеся подмножества кодов ошибок в рамках множества кодов ошибок, которые соответствуют одному глобальному идентификатору. Например, множество кодов ошибок с глобальным идентификатором `RC_SPACE_DRIVERS` включает непересекающиеся подмножества кодов ошибок с локальными идентификаторами `RC_FACILITY_I2C`, `RC_FACILITY_USB`, `RC_FACILITY_BLKDEV`.

Глобальные и локальные идентификаторы специфичных кодов ошибок назначаются разработчиками решения на базе KasperskyOS и разработчиками ПО из состава KasperskyOS SDK независимо друг от друга. То есть формируется два множества глобальных идентификаторов. Каждый глобальный идентификатор имеет уникальное смысловое значение в рамках одного множества. Каждый локальный идентификатор имеет уникальное смысловое значение в рамках множества локальных идентификаторов, относящихся к одному глобальному идентификатору. Общие коды ошибок могут использоваться в любых API.

Такой централизованный подход позволяет избежать появления в решении на базе KasperskyOS одинаковых кодов ошибок с разными смысловыми значениями. Это нужно, чтобы исключить проблему транзита кодов ошибок через разные API. Например, такая проблема возникает, когда драйверы вызывают функции библиотеки `kdf`, получают коды ошибок и возвращают эти коды через свои API. Если формировать коды ошибок без централизованного подхода, то один и тот же код ошибки может иметь разные смысловые значения для библиотеки `kdf` и для драйвера. В таких условиях драйверы возвращают корректные коды ошибок, если только выполняется преобразование кодов ошибок библиотеки `kdf` в коды ошибок каждого из драйверов. То есть коды ошибок в решении на базе KasperskyOS назначаются так, чтобы не выполнять конвертацию этих кодов при транзите через разные API.

> Приведенные здесь сведения о кодах возврата не относятся к функциям интерфейса POSIX и API стороннего ПО, используемого в решениях на базе KasperskyOS.

Общие коды возврата

Коды возврата, которые являются общими для API любых компонентов решения и их составных частей, определены в заголовочном файле `sysroot-*-kos/include/rtl/retcode.h` из состава KasperskyOS SDK. Описание общих кодов возврата приведено в таблице ниже.

Общие коды возврата

Код возврата	Описание
`rcOk` (соответствует значению `0`)	Функция завершилась успешно.
`rcInvalidArgument`	Параметр функции некорректен.
`rcNotConnected`	Нет соединения между клиентской и серверной сторонами взаимодействия. Например, отсутствует серверный IPC-дескриптор.
`rcOutOfMemory`	Недостаточно памяти для выполнения операции.
`rcBufferTooSmall`	Недостаточный размер буфера.
`rcInternalError`	Функция завершилась с внутренней ошибкой, которая связана с некорректной логикой. Примерами внутренних ошибок являются: выход значений за допустимые пределы, появление нулевых указателей и значений там, где этого быть не должно.
`rcTransferError`	Ошибка отправки IPC-сообщения.
`rcReceiveError`	Ошибка приема IPC-сообщения.
`rcSourceFault`	IPC-сообщение не было передано из-за источника IPC-сообщения.
`rcTargetFault`	IPC-сообщение не было передано из-за приемника IPC-сообщения.
`rcIpcInterrupt`	IPC прервано другим потоком процесса.
`rcRestart`	Сигнализирует, что функцию нужно вызвать повторно.
`rcFail`	Функция завершилась с ошибкой.
`rcNoCapability`	Операция над ресурсом недоступна.
`rcNotReady`	Инициализация не выполнена.
`rcUnimplemented`	Функция не реализована.

`rcBufferTooLarge`	Большой размер буфера.
`rcBusy`	Ресурс временно недоступен.
`rcResourceNotFound`	Ресурс не найден.
`rcTimeout`	Время ожидания истекло.
`rcSecurityDisallow`	Операция запрещена механизмами безопасности.
`rcFutexWouldBlock`	Операция приведет к блокировке.
`rcAbort`	Операция прервана.
`rcInvalidThreadState`	В обработчике прерывания вызвана недопустимая функция.
`rcAlreadyExists`	Множество элементов уже содержит добавляемый элемент.
`rcInvalidOperation`	Операция не может быть выполнена.
`rcHandleRevoked`	Права доступа к ресурсу отозваны.
`rcQuotaExceeded`	Квота на ресурс превышена.
`rcDeviceNotFound`	Устройство не найдено.
`rcOverflow`	Произошло переполнение.
`rcAlreadyDone`	Операция уже выполнена.

Определение кодов ошибок

Чтобы определить код ошибки, разработчику решения на базе KasperskyOS нужно использовать макрос `MAKE_RETCODE()`, определенный в заголовочном файле `sysroot-*-kos/include/rtl/retcode.h` из состава KasperskyOS SDK. При этом через параметр `customer` нужно передать символьную константу `RC_CUSTOMER_TRUE`.

Пример:

```
#define LV_EBADREQUEST MAKE_RETCODE(RC_CUSTOMER_TRUE, RC_SPACE_APPS,
RC_FACILITY_LogViewer, 5, "Bad request")
```

> Описание ошибки, которое передается через параметр `desc`, не используется макросом `MAKE_RETCODE()`. Это описание требуется, чтобы создать базу данных кодов ошибок при сборке решения на базе KasperskyOS. В настоящее время механизм для создания и использования такой базы данных не реализован.

Чтение полей структуры кода ошибки

Макросы `RC_GET_CUSTOMER()`, `RC_GET_SPACE()`, `RC_GET_FACILITY()` и `RC_GET_CODE()`, определенные в заголовочном файле `sysroot-*-kos/include/rtl/retcode.h` из состава KasperskyOS SDK, позволяют читать поля структуры кода ошибки.

Макросы `RETCODE_HR_PARAMS()` и `RETCODE_HR_FMT()`, определенные в заголовочном файле `sysroot-*-kos/include/rtl/retcode_hr.h` из состава KasperskyOS SDK, используются для форматированного вывода сведений об ошибке.

Библиотека libkos

Библиотека `libkos` является базовой библиотекой KasperskyOS, предоставляющей набор API, через которые программы и другие библиотеки (например, `libc`, `kdf`) используют службы ядра. API, предоставляемые библиотекой `libkos`, обеспечивают для разработчиков решения следующие возможности:

- управление процессами, потоками исполнения, виртуальной памятью;
- управление доступом к ресурсам;
- осуществление ввода-вывода;
- создание IPC-каналов;
- управление электропитанием;
- получение статистических сведений о системе;
- другие возможности, поддерживаемые службами ядра.

Этот раздел содержит подробные описания работы с некоторыми интерфейсами библиотеки `libkos`. Описания остальных интерфейсов вы можете найти в соответствующих им заголовочных файлах.

Заголовочные файлы, определяющие API библиотеки `libkos`, расположены в следующих директориях:

- `sysroot-*-kos/include/coresrv/`
- `sysroot-*-kos/include/kos/`

Управление дескрипторами (handle_api.h)

API определен в заголовочном файле `sysroot-*-kos/include/coresrv/handle/handle_api.h` из состава KasperskyOS SDK.

API предназначен для выполнения операций с дескрипторами. Дескрипторы имеют тип `Handle`, который определен в заголовочном файле `sysroot-*-kos/include/handle/handletype.h` из состава KasperskyOS SDK.

Локальность дескрипторов

Каждый процесс получает дескрипторы из своего пространства дескрипторов независимо от других процессов. Пространства дескрипторов разных процессов абсолютно идентичны, то есть представляют собой одно и то же множество значений. Поэтому дескриптор является уникальным (имеет уникальное значение) только в рамках пространства дескрипторов одного процесса, который владеет этим дескриптором. То есть разные процессы могут иметь: одинаковые дескрипторы, которые идентифицируют разные ресурсы, или разные дескрипторы, которые идентифицируют один и тот же ресурс.

Маска прав дескриптора

Маска прав дескриптора имеет размер 32 бита и состоит из общей и специальной части. Общая часть описывает права, неспецифичные для любых ресурсов (флаги этих прав определены в заголовочном файле `sysroot-*-kos/include/services/ocap.h` из состава KasperskyOS SDK). Например, в общей части находится флаг `OCAP_HANDLE_TRANSFER`, который определяет право на передачу дескриптора. Специальная часть описывает права, специфичные для пользовательского или системного ресурса. Флаги прав специальной части для системных ресурсов определены в заголовочном файле `ocap.h`. Структура специальной части для пользовательских ресурсов определяется поставщиком ресурсов с использованием макроса `OCAP_HANDLE_SPEC()`, который определен в заголовочном файле `ocap.h`. Поставщику ресурсов необходимо экспортировать публичные заголовочные файлы с описанием флагов специальной части.

При создании дескриптора системного ресурса маска прав задается ядром KasperskyOS, которое применяет маски прав из заголовочного файла `ocap.h`. Применяются маски прав с именами вида `OCAP_*_FULL` (например, `OCAP_IOPORT_FULL`, `OCAP_TASK_FULL`, `OCAP_FILE_FULL`) и вида `OCAP_IPC_*` (например, `OCAP_IPC_SERVER`, `OCAP_IPC_LISTENER`, `OCAP_IPC_CLIENT`).

При создании дескриптора пользовательского ресурса маска прав задается пользователем.

При передаче дескриптора маска прав задается пользователем, но передаваемые права доступа не могут быть повышены относительно прав доступа, которые имеет процесс.

Создание дескрипторов

Создание дескрипторов системных ресурсов

Дескрипторы системных ресурсов создаются при создании этих ресурсов, например, при регистрации прерывания или региона памяти MMIO, создании буфера DMA, потока исполнения или процесса.

Создание дескрипторов пользовательских ресурсов

Дескрипторы пользовательских ресурсов создаются поставщиками ресурсов с использованием функции `KnHandleCreateUserObject()` или `KnHandleCreateUserObjectEx()`.

Через параметр `context` нужно задать контекст пользовательского ресурса. *Контекст пользовательского ресурса* – данные, позволяющие поставщику ресурса идентифицировать ресурс и его состояние, когда запрашивается доступ к ресурсу другими процессами. В общем случае это набор разнотипных данных (структура). Например, для файла контекст может включать имя, путь, положение курсора. Контекст пользовательского ресурса используется в качестве контекста передачи ресурса или совместно с несколькими контекстами передачи ресурса.

Через параметр `rights` нужно задать маску прав дескриптора.

Создание IPC-дескрипторов

IPC-дескриптор (англ. IPC handle) – это дескриптор, который идентифицирует IPC-канал. IPC-дескрипторы используются для выполнения системных вызовов. Клиентский IPC-дескриптор нужен для выполнения системного вызова `Call()`. Серверный IPC-дескриптор требуется для выполнения системных вызовов `Recv()` и `Reply()`. Серверный IPC-дескриптор, который имеет расширенные права, позволяющие добавлять IPC-каналы в набор идентифицируемых этим дескриптором IPC-каналов, называется *слушающим дескриптором* (англ. listener handle). Клиентский IPC-дескриптор, который идентифицирует одновременно IPC-канал до сервера и службу этого сервера, называется *callable-дескриптором* (англ. callable handle).

Сервер создает callable-дескриптор и передает его клиенту, чтобы клиент мог использовать службу сервера. Клиент инициализирует IPC-транспорт, используя полученный callable-дескриптор. При этом в функции инициализации прокси-объекта клиент указывает значение `INVALID_RIID` в качестве идентификатор службы (RIID). Чтобы создать callable-дескриптор, нужно вызвать функцию `KnHandleCreateUserObjectEx()`, указав в параметрах `ipcChannel` и `riid` серверный IPC-дескриптор и идентификатор службы (RIID) соответственно. Через параметр `context` можно задать данные для ассоциации с callable-дескриптором. Сервер сможет получить указатель на эти данные при разыменовании callable-дескриптора. (Несмотря на то что callable-дескриптор является IPC-дескриптором, он помещается ядром в поле `base_.self` фиксированной части IPC-запроса.)

Чтобы создать и ассоциировать между собой клиентский, серверный и слушающий IPC-дескрипторы, нужно вызвать функцию `KnHandleConnect()` или `KnHandleConnectEx()`. Эти функции используются для статического создания IPC-каналов. Функция `KnHandleConnect()` создает IPC-дескрипторы из пространства дескрипторов вызывающего процесса. При этом клиентский IPC-дескриптор может быть передан другому процессу. Функция `KnHandleConnectEx()` может создать IPC-дескрипторы как из пространства дескрипторов вызывающего процесса, так и из пространств дескрипторов других процессов: клиента и сервера.

При вызове функции `KnHandleConnect()` или `KnHandleConnectEx()` со значением `INVALID_HANDLE` в параметре, задающем слушающий дескриптор, создается новый слушающий дескриптор. При этом серверный и слушающий IPC-дескрипторы в выходных параметрах являются одним и тем же дескриптором. Если вызвать функцию `KnHandleConnect()` или `KnHandleConnectEx()`, указав слушающий дескриптор, то созданный серверный IPC-дескриптор обеспечит возможность получать IPC-запросы по всем IPC-каналам, ассоциированным с этим слушающим дескриптором. В этом случае серверный и слушающий IPC-дескрипторы в выходных параметрах являются разными дескрипторами. (Первый IPC-канал, ассоциированный со слушающим дескриптором, создается при вызове функции `KnHandleConnect()` или `KnHandleConnectEx()` со значением `INVALID_HANDLE` в параметре, задающем слушающий дескриптор. Второй и последующие IPC-каналы, ассоциированные со слушающим дескриптором, создаются при втором и последующих вызовах функции `KnHandleConnect()` или `KnHandleConnectEx()` с указанием слушающего дескриптора, полученного при первом вызове.)

Чтобы создать слушающий дескриптор, не связанный с клиентским и серверным IPC-дескрипторами, нужно вызвать функцию `KnHandleCreateListener()`. (Функции `KnHandleConnect()` и `KnHandleConnectEx()` создают слушающий дескриптор, связанный с клиентским и серверным IPC-дескрипторами.) Функцию `KnHandleCreateListener()` удобно использовать, чтобы создать слушающий дескриптор, с которым впоследствии будут связаны callable-дескрипторы.

Чтобы создать клиентский IPC-дескриптор для обращения к модулю безопасности Kaspersky Security Module через интерфейс безопасности, нужно вызвать функцию `KnHandleSecurityConnect()`. Эта функция вызывается библиотекой `libkos` при инициализации IPC-транспорта для обращения к модулю безопасности.

Сведения о функциях API

Функции handle_api.h

Функция	Сведения о функции
`KnHandleCreateUserObject()`	Назначение

	Создает дескриптор. Параметры • [in] `type` – тип дескриптора. Фиктивный параметр, который должен принимать значение от константы `HANDLE_TYPE_USER_FIRST` до константы `HANDLE_TYPE_USER_LAST`, определенных в заголовочном файле `sysroot-*-kos/include/handle/handletype.h` из состава KasperskyOS SDK. • [in] `rights` – маска прав дескриптора. • [in,optional] `context` – указатель на данные, которые нужно ассоциировать с дескриптором, или `RTL_NULL`, если эта ассоциация не требуется. • [out] `handle` – указатель на дескриптор. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.
`KnHandleCreateUserObjectEx()`	Назначение Создает дескриптор. Параметры • [in] `type` – тип дескриптора. Фиктивный параметр, который должен принимать значение от константы `HANDLE_TYPE_USER_FIRST` до константы `HANDLE_TYPE_USER_LAST`, определенных в заголовочном файле `sysroot-*-kos/include/handle/handletype.h` из состава KasperskyOS SDK. • [in] `rights` – маска прав дескриптора. • [in,optional] `context` – указатель на данные, которые нужно ассоциировать с дескриптором, или `RTL_NULL`, если эта ассоциация не требуется. • [in,optional] `ipcChannel` – серверный IPC-дескриптор или `INVALID_HANDLE`, если не требуется создавать callable-дескриптор. • [in,optional] `riid` – идентификатор службы (RIID) или `INVALID_RIID`, если не требуется создавать callable-дескриптор. • [out] `handle` – указатель на дескриптор. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.
`KnHandleConnect()`	Назначение

	Создает и связывает между собой клиентский, серверный и слушающий IPC-дескрипторы. Параметры • [in,optional] `ls` – слушающий дескриптор или `INVALID_HANDLE`, чтобы создать его. • [out,optional] `outLs` – указатель на слушающий дескриптор. Можно указать `RTL_NULL`, если через параметр `ls` задан слушающий дескриптор. • [out,optional] `outSr` – указатель на серверный IPC-дескриптор или `RTL_NULL`, чтобы не создавать серверный IPC-дескриптор, если через параметр `ls` задан слушающий дескриптор. • [out] `outCl` – указатель на клиентский IPC-дескриптор. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.
`KnHandleConnectEx()`	Назначение Создает и связывает между собой клиентский, серверный и слушающий IPC-дескрипторы. Параметры • [in] `server` – дескриптор серверного процесса. • [in,optional] `srListener` – слушающий дескриптор из пространства дескрипторов серверного процесса или `INVALID_HANDLE`, чтобы создать его. • [in] `client` – дескриптор клиентского процесса. • [out,optional] `outSrListener` – указатель на слушающий дескриптор из пространства дескрипторов серверного процесса. Можно указать `RTL_NULL`, если через параметр `srListener` задан слушающий дескриптор. • [out,optional] `outSrEndpoint` – указатель на серверный IPC-дескриптор из пространства дескрипторов серверного процесса или `RTL_NULL`, чтобы не создавать серверный IPC-дескриптор, если через параметр `srListener` задан слушающий дескриптор. • [out] `outClEndpoint` – указатель на клиентский IPC-дескриптор из пространства дескрипторов клиентского процесса. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.

`KnHandleSecurityConnect()`	Назначение Создает клиентский IPC-дескриптор для обращения к модулю безопасности Kaspersky Security Module через интерфейс безопасности. Параметры • [out] `client` – указатель на дескриптор. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.
`KnHandleCreateListener()`	Назначение Создает слушающий дескриптор, не связанный с клиентским и серверным IPC-дескрипторами. Параметры • [out] `listener` – указатель на слушающий дескриптор. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.

Передача дескрипторов

Общие сведения

Передача дескрипторов между процессами осуществляется, чтобы потребители ресурсов получили доступ к требуемым ресурсам. По причине локальности дескрипторов передача дескриптора инициирует на стороне принимающего процесса создание дескриптора из его пространства дескрипторов. Этот дескриптор регистрируется как потомок отправленного дескриптора и идентифицирует тот же ресурс.

Один дескриптор может быть передан многократно одному или нескольким процессам. Каждая передача порождает нового потомка переданного дескриптора на стороне принимающего процесса. Процесс может передавать дескрипторы, которые он получил от других процессов или ядра KasperskyOS. Поэтому у дескриптора может быть несколько поколений потомков. Иерархия порождения дескрипторов для каждого ресурса хранится в ядре KasperskyOS в виде *дерева наследования дескрипторов*.

Процесс может передавать дескрипторы как пользовательских, так и системных ресурсов, если права доступа этих дескрипторов разрешают выполнять передачу (в маске прав установлен флаг `OCAP_HANDLE_TRANSFER`). У потомка может быть меньше прав доступа, чем у предка. Например, передающий процесс имеет права доступа к файлу на чтение и запись, а передает права доступа только на чтение. Передающий процесс также может запретить принимающему процессу дальнейшую передачу дескриптора. Права доступа задаются в передаваемой маске прав дескриптора.

Условия для передачи дескрипторов

Чтобы процессы могли передавать дескрипторы между собой, должны выполняться следующие условия:

1. Между процессами создан IPC-канал.

2. Политика безопасности решения (`security.psl`) разрешает взаимодействие классов процессов.

3. Реализованы интерфейсные методы для передачи дескрипторов.

> API task.h позволяет родительскому процессу передавать дескрипторы дочернему процессу, который еще не запущен.

В IDL-описании сигнатуры интерфейсных методов для передачи дескрипторов имеют входные (`in`) и/или выходные (`out`) параметры типа `Handle` или `array` с элементами типа `Handle`. Через входные параметры одного метода можно передать до 255 дескрипторов. Столько же дескрипторов можно получить через выходные параметры.

Пример IDL-описания, где заданы сигнатуры интерфейсных методов для передачи дескрипторов:

```
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);
}
```

Для каждого параметра типа `Handle` компилятор NK генерирует в структурах IPC-запросов `*_req` и/или IPC-ответов `*_res` поле типа `nk_handle_desc_t` (далее также *транспортный контейнер дескриптора*). Этот тип объявлен в заголовочном файле `sysroot-*-kos/include/nk/types.h` из состава KasperskyOS SDK и представляет собой структуру, состоящую из трех полей: поля дескриптора `handle`, поля маски прав дескриптора `rights` и поля контекста передачи ресурса `badge`.

Контекст передачи ресурса

Контекст передачи ресурса – данные, позволяющие серверу идентифицировать ресурс и его состояние, когда запрашивается доступ к ресурсу через потомков переданного дескриптора. В общем случае это набор разнотипных данных (структура). Например, для файла контекст передачи может включать имя, путь, положение курсора. Сервер получает указатель на контекст передачи ресурса при разыменовании дескриптора.

Сервер независимо от того, является ли она поставщиком ресурса или нет, может ассоциировать каждую передачу дескриптора с отдельным контекстом передачи ресурса. Этот контекст передачи ресурса связывается только с теми потомками дескриптора (поддеревом наследования дескриптора), которые порождены в результате конкретной его передачи. Это позволяет определять состояние ресурса по отношению к отдельной передаче дескриптора этого ресурса. Например, в случае множественного доступа к одному файлу контекст передачи файла позволяет определить, какому именно открытию этого файла соответствует полученный IPC-запрос.

Если сервер является поставщиком ресурса, то по умолчанию каждая передача дескриптора этого ресурса ассоциируется с контекстом пользовательского ресурса. То есть контекст пользовательского ресурса используется в качестве контекста передачи ресурса для каждой передачи дескриптора, если эта передача не ассоциируется с отдельным контекстом передачи ресурса.

Сервер, который является поставщиком ресурса, может использовать совместно контекст пользовательского ресурса и контексты передачи ресурса. Например, имя, путь и размер файла хранятся в контексте пользовательского ресурса, а положение курсора хранится в нескольких контекстах передачи ресурса, так как каждый клиент может работать с разными частями файла. Технически совместное использование контекста пользовательского ресурса и контекстов передачи ресурса достигается тем, что контексты передачи ресурса хранят указатель на контекст пользовательского ресурса.

Если клиент использует несколько разнотипных ресурсов сервера, контексты передачи ресурсов (или контексты пользовательских ресурсов, если они используются в качестве контекстов передачи ресурсов) должны быть типизированными объектами `KosObject`. Это нужно, чтобы сервер мог проверить, что клиент при использовании ресурса передал в интерфейсный метод дескриптор того ресурса, который соответствует этому методу. Такая проверка требуется, поскольку клиент может ошибочно передать в интерфейсный метод дескриптор ресурса, который не соответствует этому методу. Например, клиент получил дескриптор файла и передал его в интерфейсный метод для работы с томами.

Чтобы ассоциировать передачу дескриптора с контекстом передачи ресурса, сервер помещает в поле `badge` структуры `nk_handle_desc_t` дескриптор объекта контекста передачи ресурса. *Объект контекста передачи ресурса* – объект ядра, в котором хранится указатель на контекст передачи ресурса. Чтобы создать объект контекста передачи ресурса, нужно вызвать функцию `KnHandleCreateBadge()`. Работа этой функции связана с механизмом уведомлений, так как серверу нужно знать, когда объект контекста передачи ресурса будет закрыт и удален. Эти сведения требуются серверу, чтобы освободить или использовать повторно память, которая отведена для хранения контекста передачи ресурса.

Объект контекста передачи ресурса будет закрыт, когда будут закрыты или отозваны потомки дескриптора, которые образуют поддерево наследования дескрипторов, корневой узел которого порожден передачей этого дескриптора в ассоциации с этим объектом. (Переданный дескриптор может быть закрыт не только целенаправленно, но и, например, при неожиданном завершении работы принимающего клиента.) Получив уведомление о закрытии объекта контекста передачи ресурса, сервер закрывает дескриптор этого объекта. После этого объект контекста передачи ресурса будет удален. Получив уведомление, что объект контекста передачи ресурса удален, сервер освобождает или использует повторно память, которая отведена для хранения контекста передачи ресурса.

Один объект контекста передачи ресурса может быть ассоциирован только с одной передачей дескриптора.

Упаковка данных в транспортный контейнер дескриптора

Чтобы упаковать дескриптор, маску прав дескриптора и дескриптор объекта контекста передачи ресурса в транспортный контейнер дескриптора, нужно использовать макрос `nk_handle_desc()`, который определен в заголовочном файле `sysroot-*-kos/include/nk/types.h` из состава KasperskyOS SDK. Этот макрос принимает переменное число параметров.

Если не передавать макросу ни одного параметра, то в поле дескриптора `handle` структуры `nk_handle_desc_t` будет записано значение `NK_INVALID_HANDLE`. Если передать макросу один параметр, то этот параметр интерпретируется как дескриптор. Если передать макросу два параметра, то первый параметр интерпретируется как дескриптор, второй параметр интерпретируется как маска прав дескриптора. Если передать макросу три параметра, то первый параметр интерпретируется как дескриптор, второй параметр интерпретируется как маска прав дескриптора, третий параметр интерпретируется как дескриптор объекта контекста передачи ресурса.

Извлечение данных из транспортного контейнера дескриптора

Чтобы извлечь дескриптор, маску прав дескриптора и указатель на контекст передачи ресурса из транспортного контейнера дескриптора, нужно использовать соответственно функции `nk_get_handle()`, `nk_get_rights()` и `nk_get_badge_op()` (или `nk_get_badge()`), которые определены в заголовочном файле `sysroot-*-kos/include/nk/types.h` из состава KasperskyOS SDK. Функции `nk_get_badge_op()` и `nk_get_badge()` нужно использовать только при разыменовании дескрипторов.

Сценарии передачи дескрипторов

Сценарий передачи дескрипторов от клиента к серверу включает следующие шаги:

1. Клиент упаковывает дескрипторы и маски прав дескрипторов в поля структуры IPC-запросов `*_req` типа `nk_handle_desc_t`.
2. Клиент вызывает интерфейсный метод для передачи дескрипторов серверу. При вызове этого метода выполняется системный вызов `Call()`.
3. Сервер получает IPC-запрос, выполняя системный вызов `Recv()`.
4. Диспетчер на стороне сервера вызывает метод, который соответствует IPC-запросу. Этот метод извлекает дескрипторы и маски прав дескрипторов из полей структуры IPC-запросов `*_req` типа `nk_handle_desc_t`.

Сценарий передачи дескрипторов от сервера к клиенту включает следующие шаги:

1. Клиент вызывает интерфейсный метод для получения дескрипторов от сервера. При вызове этого метода выполняется системный вызов `Call()`.
2. Сервер получает IPC-запрос, выполняя системный вызов `Recv()`.
3. Диспетчер на стороне сервера вызывает метод, который соответствует IPC-запросу. Этот метод упаковывает дескрипторы, маски прав дескрипторов и дескрипторы объектов контекстов передачи ресурсов в поля структуры IPC-ответов `*_res` типа `nk_handle_desc_t`.
4. Сервер отвечает на IPC-запрос, выполняя системный вызов `Reply()`.
5. На стороне клиента интерфейсный метод возвращает управление. После этого клиент извлекает дескрипторы и маски прав дескрипторов из полей структуры IPC-ответов `*_res` типа `nk_handle_desc_t`.

Если передающий процесс задает в передаваемой маске прав дескриптора больше прав доступа, чем задано для передаваемого дескриптора (владельцем которого он является), то передача не осуществляется. В этом случае выполнение системного вызова `Call()` передающим или принимающим клиентом, а также выполнение системного вызова `Reply()` передающим сервером завершается с ошибкой `rcSecurityDisallow`.

Сведения о функциях API

Функции handle_api.h

Функция	Сведения о функции
`KnHandleCreateBadge()`	Назначение Создает объект контекста передачи ресурса и настраивает механизм уведомлений для контроля жизненного цикла этого объекта.

	Параметры • [in] `notice` – идентификатор приемника уведомлений. • [in] `eventId` – идентификатор записи вида "ресурс – маска событий" в приемнике уведомлений. • [in,optional] `context` – указатель на данные, которые нужно ассоциировать с передачей дескриптора, или `RTL_NULL`, если эта ассоциация не требуется. • [out] `handle` – указатель на дескриптор объекта контекста передачи ресурса. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки. Дополнительные сведения Приемник уведомлений настраивается на получение уведомлений о событиях, которые соответствуют флагам маски событий `EVENT_OBJECT_DESTROYED` и `EVENT_BADGE_CLOSED`.

Копирование дескрипторов

Копирование дескриптора представляет собой операцию, похожую на передачу дескриптора, но выполняемую внутри процесса. Потомок дескриптора создается в том же процессе и из того же пространства дескрипторов. Права потомка дескриптора могут быть меньше либо равны правам оригинального дескриптора. Копирование дескриптора можно ассоциировать с объектом контекста передачи ресурса. Это позволит отследить через механизм уведомлений, когда будут закрыты или отозваны все потомки дескриптора, образующие поддерево наследования дескрипторов, корневой узел которого порожден копированием. Также это обеспечит возможность отозвать этих потомков.

Чтобы выполнить копирование дескриптора, нужно вызвать функцию `KnHandleCopy()`. При этом в маске прав дескриптора должен быть установлен флаг `OCAP_HANDLE_COPY`.

Сведения о функциях API приведены в таблице ниже.

Функции handle_api.h

Функция	Сведения о функции
`KnHandleCopy()`	Назначение Копирует дескриптор. В результате копирования вызывающий процесс получает потомка дескриптора. Параметры • [in] `inHandle` – оригинальный дескриптор. • [in] `newRightsMask` – маска прав потомка дескриптора.

	• [in,optional] `copyBadge` – дескриптор объекта контекста передачи ресурса или `INVALID_HANDLE`, если не требуется ассоциировать копирование дескриптора с этим объектом. • [out] `outHandle` – указатель на потомка дескриптора. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.

Разыменование дескрипторов

Разыменование дескриптора – это операция, при которой клиент отправляет серверу дескриптор, а сервер получает указатель на контекст передачи ресурса, маску прав отправленного дескриптора и предка отправленного клиентом дескриптора, которым сервер уже владеет. Разыменование выполняется, когда потребитель ресурсов, вызывая методы работы с ресурсом (например, чтение, запись, закрытие доступа), передает поставщику ресурсов дескриптор, который был получен от этого поставщика ресурсов при открытии доступа к ресурсу.

Разыменование дескрипторов требует выполнения тех же условий и использует те же механизмы и типы данных, что и передача дескрипторов. Сценарий разыменования дескриптора включает следующие шаги:

1. Клиент упаковывает дескриптор в поле структуры IPC-запросов `*_req` типа `nk_handle_desc_t`.
2. Клиент вызывает интерфейсный метод для отправки дескриптора серверу с целью выполнения действий с ресурсом. При вызове этого метода выполняется системный вызов `Call()`.
3. Сервер принимает IPC-запрос, выполнив системный вызов `Recv()`.
4. Диспетчер на стороне сервера вызывает метод, который соответствует IPC-запросу. Этот метод проверяет, что выполнена именно операция разыменования, а не передача дескриптора. Затем вызванный метод опционально проверяет, что права доступа разыменованного дескриптора (который отправлен клиентом) разрешают запрашиваемые действия с ресурсом, и извлекает указатель на контекст передачи ресурса из поля структуры запросов `*_req` типа `nk_handle_desc_t`.

Для выполнения проверок сервер использует функции `nk_is_handle_dereferenced()` и `nk_get_badge_op()`, которые объявлены в заголовочном файле `sysroot-*-kos/include/nk/types.h` из состава KasperskyOS SDK.

types.h (фрагмент)

```
/**
 * Функция возвращает отличное от нуля значение, если
 * дескриптор в транспортном контейнере дескриптора
 * desc получен в результате операции разыменования
 * дескриптора. Функция возвращает нуль, если дескриптор
 * в транспортном контейнере дескриптора desc получен
 * в результате операции передачи дескриптора.
 */
static inline
nk_bool_t nk_is_handle_dereferenced(const nk_handle_desc_t *desc)

/**
 * Функция извлекает указатель на контекст передачи ресурса
 * badge из транспортного контейнера дескриптора desc,
```

```
 * если в маске прав, которая помещена в транспортном
 * контейнере дескриптора desc, установлены флаги operation.
 * В случае успеха функция возвращает NK_EOK, иначе возвращает код ошибки.
 */
static inline
nk_err_t nk_get_badge_op(const nk_handle_desc_t *desc,
                         nk_rights_t operation,
                         nk_badge_t *badge)
```

В общем случае серверу не требуется дескриптор, который получен в результате разыменования, поскольку сервер, как правило, сохраняет дескрипторы, которыми владеет, например, в составе контекстов пользовательских ресурсов. Но при необходимости сервер может извлечь этот дескриптор из транспортного контейнера дескриптора.

Отзыв дескрипторов

Процесс может отозвать потомков дескриптора, которым он владеет. Отзыв дескрипторов осуществляется на основе дерева наследования дескрипторов.

Отзыв дескрипторов не закрывает их, но через отозванные дескрипторы невозможно обращаться к ресурсам. Любая функция, которая принимает дескриптор, завершается с ошибкой `rcHandleRevoked`, если эта функция вызвана с отозванным дескриптором.

Чтобы отозвать потомков дескриптора, нужно вызвать функцию `KnHandleRevoke()` или `KnHandleRevokeSubtree()`. Функция `KnHandleRevokeSubtree()` использует объект контекста передачи ресурса, который создается при передаче дескрипторов.

Если каждый из дескрипторов системного ресурса во всех процессах, которые владеют этими дескрипторами, будет закрыт (см. "Закрытие дескрипторов") или отозван, то этот системный ресурс будет удален.

Сведения о функциях API приведены в таблице ниже.

Функции handle_api.h

Функция	Сведения о функции
`KnHandleRevoke()`	Назначение Закрывает дескриптор и отзывает его потомков. Параметры • [in] `handle` – дескриптор. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.
`KnHandleRevokeSubtree()`	Назначение Отзывает дескрипторы, которые образуют поддерево наследования заданного дескриптора. Параметры

	• [in] `handle` – дескриптор. Дескрипторы, образующие поддерево наследования этого дескриптора, отзываются. • [in] `badge` – дескриптор, идентифицирующий объект контекста передачи ресурса, который определяет поддерево наследования дескрипторов для отзыва. Корневым узлом этого поддерева является дескриптор, который порожден передачей или копированием дескриптора, заданного через параметр `handle`, в ассоциации с объектом контекста передачи ресурса. Возвращаемые значения В случае успеха возвращает `rc0k`, иначе возвращает код ошибки.

Закрытие дескрипторов

Процесс может закрыть дескрипторы, которыми он владеет. Закрытие дескриптора прекращает ассоциацию идентификатора с ресурсом, тем самым освобождая идентификатор. Закрытие дескриптора не делает недействительными его предков и потомков (в отличие от отзыва дескриптора, который делает недействительными его потомков). То есть через предков и потомков закрытого дескриптора обеспечивается доступ к ресурсу, который они идентифицируют. Также закрытие дескриптора не нарушает целостность дерева наследования дескрипторов, которое относится к ресурсу, идентифицируемому этим дескриптором. Место закрытого дескриптора занимает его предок. То есть предок закрытого дескриптора становится непосредственным предком потомков закрытого дескриптора.

Чтобы закрыть дескриптор, нужно вызвать функцию `KnHandleClose()`.

Если каждый из дескрипторов системного ресурса во всех процессах, которые владеют этими дескрипторами, будет отозван (см. "Отзыв дескрипторов") или закрыт, то этот системный ресурс будет удален.

Сведения о функциях API приведены в таблице ниже.

Функции handle_api.h

Функция	Сведения о функции
`KnHandleClose()`	Назначение Закрывает дескриптор. Параметры • [in] `handle` – дескриптор. Возвращаемые значения В случае успеха возвращает `rc0k`, иначе возвращает код ошибки.

Получение идентификатора безопасности (SID)

Получив значения идентификаторов безопасности для разных дескрипторов, можно определить, идентифицируют ли эти дескрипторы разные ресурсы или один и тот же ресурс.

Чтобы получить идентификатор безопасности по дескриптору, нужно вызвать функцию `KnHandleGetSidByHandle()`. При этом в маске прав дескриптора должен быть установлен флаг `OCAP_HANDLE_GET_SID`.

Сведения о функциях API приведены в таблице ниже.

Функции handle_api.h

Функция	Сведения о функции
`KnHandleGetSidByHandle()`	Назначение Позволяет получить идентификатор безопасности (SID) по дескриптору. Параметры • [in] `handle` – дескриптор. • [out] `sid` – указатель на идентификатор безопасности (SID). Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.

Пример использования OCap

В этом примере приведен сценарий использования OCap, в котором поставщик ресурсов предоставляет следующие методы доступа к своим ресурсам:

- `OpenResource()` – открытие доступа к ресурсу;
- `UseResource()` – использование ресурса;
- `CloseResource()` – закрытие доступа к ресурсу.

Потребитель ресурсов использует эти методы.

IDL-описание:

```
package SimpleOCap
interface {
    OpenResource(in UInt32 ID, out Handle handle);
    UseResource(in Handle handle, in UInt8 param, out UInt8 result);
    CloseResource(in Handle handle);
}
```

Сценарий включает следующие шаги:

1. Поставщик ресурсов создает контекст пользовательского ресурса и вызывает функцию `KnHandleCreateUserObject()` для создания дескриптора ресурса. Поставщик ресурсов сохраняет дескриптор ресурса в контексте пользовательского ресурса.

2. Потребитель ресурсов вызывает метод открытия доступа к ресурсу `OpenResource()`.

a. Поставщик ресурсов создает контекст передачи ресурса и вызывает функцию `KnHandleCreateBadge()` для создания объекта контекста передачи ресурса и настройки приемника уведомлений на получение уведомлений о закрытии и удалении объекта контекста передачи ресурса. Поставщик ресурсов сохраняет дескриптор объекта контекста передачи ресурса и указатель на контекст пользовательского ресурса в контексте передачи ресурса.

b. Поставщик ресурсов, используя макрос `nk_handle_desc()`, упаковывает дескриптор ресурса, маску прав дескриптора и указатель на объект контекста передачи ресурса в транспортный контейнер дескриптора.

c. Выполняется передача дескриптора от поставщика ресурсов к потребителю ресурсов, в результате которой потребитель ресурсов получает потомка дескриптора, которым владеет поставщик ресурсов.

d. Вызов метода `OpenResource()` завершается успешно. Потребитель ресурсов извлекает дескриптор и маску прав дескриптора из транспортного контейнера дескриптора функциями `nk_get_handle()` и `nk_get_rights()` соответственно. Маска прав дескриптора не требуется потребителю ресурсов для обращения к ресурсу и передается, чтобы потребитель ресурсов мог узнать свои права доступа к ресурсу.

3. Потребитель ресурсов вызывает метод использования ресурса `UseResource()`.

a. Дескриптор, который получен от поставщика ресурсов на шаге 2, используется в качестве параметра метода `UseResource()`. Перед вызовом этого метода потребитель ресурсов упаковывает дескриптор в транспортный контейнер дескриптора макросом `nk_handle_desc()`.

b. Выполняется разыменование дескриптора, в результате которого поставщик ресурсов получает указатель на контекст передачи ресурса.

c. Поставщик ресурсов, используя функцию `nk_is_handle_dereferenced()`, проверяет, что выполнена операция разыменования, а не передача дескриптора.

d. Поставщик ресурсов проверяет, что права доступа разыменованного дескриптора (который отправлен потребителем ресурсов) разрешают запрашиваемую операцию с ресурсом, и извлекает указатель на контекст передачи ресурса из транспортного контейнера дескриптора. Для этого поставщик ресурсов использует функцию `nk_get_badge_op()`, которая извлекает указатель на контекст передачи ресурса из транспортного контейнера дескриптора, если в полученной маске прав установлены флаги, соответствующие запрашиваемой операции.

e. Поставщик ресурсов, используя контекст передачи ресурса и контекст пользовательского ресурса, выполняет запрашиваемую потребителем ресурсов операцию с ресурсом. Затем поставщик ресурсов отправляет потребителю ресурсов результат выполнения этой операции.

f. Вызов метода `UseResource()` завершается успешно. Потребитель ресурсов получает результат выполнения операции с ресурсом.

4. Потребитель ресурсов вызывает метод закрытия доступа к ресурсу `CloseResource()`.

a. Дескриптор, который получен от поставщика ресурсов на шаге 2, используется в качестве параметра метода `CloseResource()`. Перед вызовом этого метода потребитель ресурсов упаковывает дескриптор в транспортный контейнер дескриптора макросом `nk_handle_desc()`. После вызова метода `CloseResource()` потребитель ресурсов закрывает дескриптор функцией `KnHandleClose()`.

b. Выполняется разыменование дескриптора, в результате которого поставщик ресурсов получает указатель на контекст передачи ресурса.

c. Поставщик ресурсов, используя функцию `nk_is_handle_dereferenced()`, проверяет, что выполнена операция разыменования, а не передача дескриптора.

d. Поставщик ресурсов, используя функцию `nk_get_badge()`, извлекает указатель на контекст передачи ресурса из транспортного контейнера дескриптора.

e. Поставщик ресурсов отзывает дескриптор, которым владеет потребитель ресурсов, функцией `KnHandleRevokeSubtree()`. В качестве параметров этой функции используются дескриптор ресурса, которым владеет поставщик ресурсов, и дескриптор объекта контекста передачи ресурса. Поставщик ресурсов получает доступ к этим дескрипторам через указатель на контекст передачи ресурса. (Технически не требуется отзывать дескриптор, которым владеет потребитель ресурсов, так как потребитель ресурсов его уже закрыл. Но поставщик ресурсов не может быть уверен в том, что потребитель ресурсов закрыл дескриптор, поэтому выполняется отзыв).

f. Вызов метода `CloseResource()` завершается успешно.

5. Поставщик ресурсов освобождает память, которая была выделена для контекста передачи ресурса и контекста пользовательского ресурса.

a. Поставщик ресурсов вызовом функции `KnNoticeGetEvent()` получает уведомление, что объект контекста передачи ресурса закрыт, и закрывает дескриптор объекта контекста передачи ресурса функцией `KnHandleClose()`.

b. Поставщик ресурсов вызовом функции `KnNoticeGetEvent()` получает уведомление, что объект контекста передачи ресурса удален, и освобождает память, которая была выделена для контекста передачи ресурса.

c. Поставщик ресурсов закрывает дескриптор ресурса функцией `KnHandleClose()` и освобождает память, которая была выделена для контекста пользовательского ресурса.

Выделение и освобождение памяти (alloc.h)

API определен в заголовочном файле `sysroot-*-kos/include/kos/alloc.h` из состава KasperskyOS SDK.

API предназначен для выделения и освобождения памяти. Выделенная память представляет собой зафиксированный регион виртуальной памяти, к которому разрешен доступ на чтение и запись.

Сведения о функциях API приведены в таблице ниже.

Функции alloc.h

Функция	Сведения о функции
`KosMemAllocEx()`	Назначение Выделяет память. Параметры • [in] `size` – размер выделяемой памяти в байтах. • [in] `align` – значение, задающее выравнивание выделяемой памяти. Должно быть степенью двойки. Адрес выделяемой памяти может быть невыравненным (*align=1*) или выравненным (*align=2,4,...,2^N*) на границу *2^N*-байтовой последовательности (например, двухбайтовой, четырехбайтовой). При выравнивании адреса размер выделяемой памяти может быть увеличен до ближайшего значения, кратного *2^N*.

	• [in] `zeroed` – значение, задающее инициализацию выделяемой памяти (`1` – инициализировать нулями, `0` – не инициализировать). Возвращаемые значения В случае успеха возвращает указатель на выделенную память, иначе возвращает `RTL_NULL`.
`KosMemAlloc()`	Назначение Выделяет память. Параметры • [in] `size` – размер выделяемой памяти в байтах. Возвращаемые значения В случае успеха возвращает указатель на выделенную памяти, иначе возвращает `RTL_NULL`.
`KosMemZalloc()`	Назначение Выделяет память и инициализирует ее нулями. Параметры • [in] `size` – размер выделяемой памяти в байтах. Возвращаемые значения В случае успеха возвращает указатель на выделенную память, иначе возвращает `RTL_NULL`.
`KosMemFree()`	Назначение Освобождает память. Параметры • [in] `ptr` – указатель на освобождаемую память. Возвращаемые значения Нет.
`KosMemGetSize()`	Назначение Позволяет получить фактический размер выделенной памяти. Фактический размер выделенной памяти превышает запрошенный, так как включает размер служебных данных, а также может быть увеличен в результате выравнивания при вызове функции `KosMemAllocEx()`. Параметры • [in] `ptr` – указатель на выделенную память.

	Возвращаемые значения Фактический размер выделенной памяти в байтах.
`KosMemGetOrigSize()`	Назначение Позволяет получить размер памяти, который был запрошен при ее выделении. Фактический размер выделенной памяти превышает запрошенный, так как включает размер служебных данных, а также может быть увеличен в результате выравнивания при вызове функции `KosMemAllocEx()`. Параметры • [in] `ptr` – указатель на выделенную память. Возвращаемые значения Размер памяти, который был запрошен при ее выделении, в байтах.

Использование DMA (dma.h)

API определен в заголовочном файле `sysroot-*-kos/include/coresrv/io/dma.h` из состава KasperskyOS SDK.

API предназначен для организации обмена данными между устройствами и оперативной памятью в режиме *прямого доступа к памяти* (англ. Direct Memory Access, DMA), при котором процессор не используется.

Сведения о функциях API приведены в таблице ниже.

Использование API

Типовой сценарий использования API включает следующие шаги:

1. Создание буфера DMA.

   Буфер DMA –это буфер, состоящий из одного или нескольких регионов физической памяти (блоков), используемых для DMA. Буфер DMA, состоящий из нескольких блоков, можно использовать, если устройство поддерживает режим "scatter/gather DMA". Буфер DMA, состоящий из одного блока, можно использовать, если устройство поддерживает режим "scatter/gather DMA" или "continuous DMA". Вероятность создать буфер DMA, состоящий из одного большого блока, ниже, чем вероятность создать буфер DMA, состоящий из нескольких небольших блоков. Это особенно актуально при высокой фрагментации физической памяти.

   > Если устройство поддерживает только режим "continuous DMA", то даже при задействовании IOMMU нужно использовать буфер DMA, состоящий из одного блока.

Чтобы выполнить этот шаг, нужно вызвать функцию `KnIoDmaCreate()` или `KnIoDmaCreateContinuous()`. Функция `KnIoDmaCreateContinuous()` создает буфер DMA, состоящий из одного блока. Функция `KnIoDmaCreate()` создает буфер DMA, состоящий из одного блока, если значение *2^order* равно значению *size/размер страницы памяти*, или значение *2^order* является ближайшим большим к значению *size/размер страницы памяти* в упорядоченном по возрастанию множестве *{2^(order-1);size/размер страницы памяти;2^order}*. Если значение *size/размер страницы памяти* больше значения *2^order*, функция `KnIoDmaCreate()` может создать буфер DMA, состоящий из нескольких блоков.

Дескриптор буфера DMA можно передать другому процессу через IPC.

2. Отображение буфера DMA на память процессов.

   Один буфер DMA может быть отображен на несколько регионов виртуальной памяти одного или нескольких процессов, которые владеют дескриптором этого буфера DMA. В результате отображения процессы получают доступ к буферу DMA на чтение и/или запись.

   Чтобы зарезервировать регион виртуальной памяти и отобразить на него буфер DMA, нужно вызвать функцию `KnIoDmaMap()`.

   Дескриптор, полученный при вызове функции `KnIoDmaMap()`, нельзя передать другому процессу через IPC.

3. Открытие доступа к буферу DMA для устройства вызовом функции `KnIoDmaBegin()`.

   Вызов функции `KnIoDmaBegin()` необходим, чтобы создать объект ядра, содержащий адреса и размеры блоков, из которых состоит буфер DMA. Эти сведения необходимы устройству, чтобы использовать буфер DMA. Устройство может работать как с физическими, так и с виртуальными адресами в зависимости от того, задействован ли IOMMU. Если IOMMU задействован, объект содержит виртуальные адреса блоков. В противном случае объект содержит физические адреса блоков.

   Дескриптор, полученный при вызове функции `KnIoDmaBegin()`, нельзя передать другому процессу через IPC.

4. Получение сведений о буфере DMA.

   На этом шаге нужно получить адреса и размеры блоков из объекта ядра, созданного вызовом функции `KnIoDmaBegin()`. Полученные адреса и размеры в дальнейшем требуется передать устройству, используя, например, MMIO. После получения этих сведений устройство может записывать в буфер DMA и/или читать из него (при задействовании IOMMU устройство на шине PCIe должно быть прикреплено к домену IOMMU).

   Чтобы выполнить этот шаг, нужно вызвать функцию `KnIoDmaGetInfo()` или `KnIoDmaContinuousGetDmaAddr()`. Функция `KnIoDmaGetInfo()` позволяет получить номер страницы памяти (`frame`) и порядок (`order`) для каждого блока. (Номер страницы памяти, умноженный на размер страницы памяти, дает адрес блока. Значение *2^order* представляет собой размер блока в страницах памяти.) Функцию `KnIoDmaContinuousGetDmaAddr()` можно использовать, если буфер DMA состоит из одного блока. Эта функция позволяет получить адрес блока. (В качестве размера блока нужно принять размер буфера DMA, заданный при его создании.)

Закрытие доступа к буферу DMA для устройства

Если удалить объект ядра, созданный при вызове функции `KnIoDmaBegin()`, то при задействовании IOMMU доступ устройства к буферу DMA будет запрещен. Чтобы удалить этот объект, нужно вызвать функцию `KnHandleClose()`, указав дескриптор, который был получен при вызове функции `KnIoDmaBegin()`. (Функция `KnHandleClose()` объявлена в заголовочном файле `sysroot-*-kos/include/coresrv/handle/handle_api.h` из состава KasperskyOS SDK.)

Удаление буфера DMA

Чтобы удалить буфер DMA, нужно выполнить следующие шаги:

1. Освободить регионы виртуальной памяти, зарезервированные при вызовах функции `KnIoDmaMap()`.

   Чтобы выполнить этот шаг, нужно использовать функцию `KnHandleClose()`, указывая дескрипторы, которые были получены при вызовах функции `KnIoDmaMap()`. (Функция `KnHandleClose()` объявлена в заголовочном файле `sysroot-*-kos/include/coresrv/handle/handle_api.h` из состава KasperskyOS SDK.)

   Этот шаг нужно выполнить для всех процессов, на память которых отображен буфер DMA.

2. Удалить объект ядра, созданный вызовом функции `KnIoDmaBegin()`.

   Чтобы выполнить этот шаг, нужно вызвать функцию `KnHandleClose()`, указав дескриптор, который был получен при вызове функции `KnIoDmaBegin()`.

3. Закрыть или отозвать каждый дескриптор буфера DMA во всех процессах, которые владеют этими дескрипторами.

   Чтобы выполнить этот шаг, нужно использовать функцию `KnHandleClose()` и/или `KnHandleRevoke()`, которые объявлены в заголовочном файле `sysroot-*-kos/include/coresrv/handle/handle_api.h` из состава KasperskyOS SDK.

Сведения о функциях API

Функции dma.h

Функция	Сведения о функции
`KnIoDmaCreate()`	Назначение Создает буфер DMA. Параметры • [in] `order` – параметр, задающий минимальное число страниц памяти (*2^order*) в блоке. • [in] `size` – размер буфера DMA в байтах. Должен быть кратен размеру страницы памяти. • [in] `flags` – флаги, задающие параметры буфера DMA. Тип параметра и флаги определены в заголовочном файле `sysroot-*-kos/include/io/io_dma.h` из состава KasperskyOS SDK. • [out] `outRid` – указатель на дескриптор буфера DMA. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки. Дополнительные сведения В параметре `flags` можно указать следующие флаги: • `DMA_DIR_TO_DEVICE` – устройство имеет доступ к буферу DMA на чтение.

	• `DMA_DIR_FROM_DEVICE` – устройство имеет доступ к буферу DMA на запись. • `DMA_DIR_BIDIR` – устройство имеет доступ к буферу DMA на чтение и запись. • `DMA_ZONE_DMA32` – для создания буфера DMA разрешено использовать только первые четыре гигабайта физической памяти. • `DMA_ATTR_WRITE_BACK`, `DMA_ATTR_WRITE_THROUGH`, `DMA_ATTR_CACHE_DISABLE`, `DMA_ATTR_WRITE_COMBINE`, `DMA_RULE_CACHE_VOLATILE`, `DMA_RULE_CACHE_FIXED` – управление кешированием.
`KnIoDmaCreateContinuous()`	Назначение Создает буфер DMA, состоящий из одного блока. Параметры • [in] `size` – размер буфера DMA в байтах. Должен быть кратен размеру страницы памяти. • [in] `flags` – флаги, задающие параметры буфера DMA. Тип параметра и флаги определены в заголовочном файле `sysroot-*-kos/include/io/io_dma.h` из состава KasperskyOS SDK. • [out] `outRid` – указатель на дескриптор буфера DMA. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки. Дополнительные сведения В параметре `flags` можно указать следующие флаги: • `DMA_DIR_TO_DEVICE` – устройство имеет доступ к буферу DMA на чтение. • `DMA_DIR_FROM_DEVICE` – устройство имеет доступ к буферу DMA на запись. • `DMA_DIR_BIDIR` – устройство имеет доступ к буферу DMA на чтение и запись. • `DMA_ZONE_DMA32` – для создания буфера DMA разрешено использовать только первые четыре гигабайта физической памяти. • `DMA_ATTR_WRITE_BACK`, `DMA_ATTR_WRITE_THROUGH`, `DMA_ATTR_CACHE_DISABLE`, `DMA_ATTR_WRITE_COMBINE`,

	`DMA_RULE_CACHE_VOLATILE`, `DMA_RULE_CACHE_FIXED` – управление кешированием.
`KnIoDmaMap()`	Назначение Резервирует регион виртуальной памяти и отображает на него буфер DMA. Параметры • [in] `rid` – дескриптор буфера DMA. • [in] `offset` – смещение в буфере DMA, с которого нужно начать отображение, в байтах. Должно быть кратно размеру страницы памяти. • [in] `length` – размер части буфера DMA, которую нужно отобразить, в байтах. Должен быть кратен размеру страницы памяти. Также должно выполняться условие: *length<=размер буфера DMA-offset*. • [in,optional] `hint` – странично выравненный желаемый базовый адрес региона виртуальной памяти или `0`, чтобы этот адрес был выбран автоматически. • [in] `vmflags` – флаги, задающие права доступа к региону виртуальной памяти. Флаги определены в заголовочном файле `sysroot-*-kos/include/vmm/flags.h` из состава KasperskyOS SDK. • [out] `addr` – базовый адрес региона виртуальной памяти. • [out] `handle` – указатель на дескриптор, который используется для освобождения региона виртуальной памяти. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки. Дополнительные сведения В параметре `vmflags` можно указать следующие флаги: • `VMM_FLAG_READ` – доступ на чтение. • `VMM_FLAG_WRITE` – доступ на запись.
`KnIoDmaModify()`	Назначение Изменяет параметры кеширования буфера DMA. Параметры

	• [in] `rid` – дескриптор буфера DMA. • [in] `newAttr` – флаги, задающие параметры кеширования буфера DMA. Флаги определены в заголовочном файле `sysroot-*-kos/include/io/io_dma.h` из состава KasperskyOS SDK. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки. Дополнительные сведения Функцию можно использовать, если выполняются следующие условия: 1. Буфер DMA был создан с указанием флага `DMA_RULE_CACHE_VOLATILE`. 2. Буфер DMA не отображен на виртуальную память. 3. Предыдущий вызов функции (если он был) был выполнен с указанием флага `DMA_RULE_CACHE_VOLATILE`. В параметре `newAttr` можно указать следующие флаги: • `DMA_ATTR_WRITE_BACK`, `DMA_ATTR_WRITE_THROUGH`, `DMA_ATTR_CACHE_DISABLE`, `DMA_ATTR_WRITE_COMBINE`, `DMA_RULE_CACHE_VOLATILE` – управление кешированием.
`KnIoDmaGetInfo()`	Назначение Позволяет получить сведения о буфере DMA. Сведения включают адреса и размеры блоков. Параметры • [in] `rid` – дескриптор буфера DMA. • [out] `outInfo` – указатель на адрес объекта, содержащего сведения о буфере DMA. Тип объекта определен в заголовочном файле `sysroot-*-kos/include/io/io_dma.h` из состава KasperskyOS SDK. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.
`KnIoDmaContinuousGetDmaAddr()`	Назначение Позволяет получить адрес блока для буфера DMA, состоящего из одного блока.

	Параметры • [in] `rid` – дескриптор буфера DMA. • [out] `addr` – адрес блока. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.
`KnIoDmaBegin()`	Назначение Открывает доступ к буферу DMA для устройства. Параметры • [in] `rid` – дескриптор буфера DMA. • [out] `handle` – указатель на дескриптор объекта ядра, содержащего адреса и размеры блоков, которые необходимы устройству, чтобы использовать буфер DMA. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.

Управление обработкой прерываний (irq.h)

API определен в заголовочном файле `sysroot-*-kos/include/coresrv/io/irq.h` из состава KasperskyOS SDK.

API позволяет управлять обработкой аппаратных прерываний. *Аппаратное прерывание* – это сигнал процессору от устройства о необходимости немедленно переключиться с исполнения текущей программы на обработку события, связанного с этим устройством. Например, нажатие клавиши на клавиатуре вызывает аппаратное прерывание, которое обеспечивает требуемую реакцию на это нажатие (к примеру, ввод символа).

Аппаратное прерывание возникает при обращении устройства к контроллеру прерываний. Это обращение может осуществляться через линию аппаратного прерывания между устройством и контроллером прерываний или через память MMIO. Во втором случае устройство выполняет запись в память MMIO, вызывая *прерывание MSI* (англ. Message Signaled Interrupt).

> В настоящее время функции для управления обработкой прерываний MSI не реализованы.

Каждой линии аппаратного прерывания соответствует одно прерывание с уникальным номером.

Сведения о функциях API приведены в таблице ниже.

Использование API

Чтобы связать прерывание с его обработчиком, нужно выполнить следующие шаги:

1. Регистрация прерывания вызовом функции `KnRegisterIrq()`.

   Одно прерывание можно зарегистрировать несколько раз в одном или нескольких процессах.

   Дескриптор прерывания можно передать другому процессу через IPC.

2. Привязка потока исполнения к прерыванию вызовом функции `KnIoAttachIrq()`.

   Этот шаг выполняется потоком исполнения, в контексте которого будет выполняться обработка прерывания.

   Используя дескриптор, полученный при вызове функции `KnRegisterIrq()`, можно привязать к прерыванию только один поток исполнения. Чтобы привязать к прерыванию несколько потоков исполнения в одном или нескольких процессах, нужно использовать разные дескрипторы этого прерывания, полученные при отдельных вызовах функции `KnRegisterIrq()`. В этом случае функцию `KnIoAttachIrq()` нужно вызывать с одними и теми же флагами в параметре `flags`.

   Дескриптор, полученный при вызове функции `KnIoAttachIrq()`, нельзя передать другому процессу через IPC.

Чтобы запретить (маскировать) прерывание, нужно вызвать функцию `KnIoDisableIrq()`. Чтобы разрешить (демаскировать) прерывание, нужно вызвать функцию `KnIoEnableIrq()`. И хотя эти функции принимают дескриптор прерывания, через который к прерыванию привязан только один поток исполнения, их действие распространяется на все потоки исполнения, привязанные к этому прерыванию. Эти функции нужно вызывать вне потоков исполнения, привязанных к прерыванию. После регистрации прерывания и привязки к нему потока исполнения это прерывание не требуется демаскировать.

Чтобы инициировать отвязывание потока исполнения от прерывания, нужно вызвать функцию `KnIoDetachIrq()` вне потока исполнения, привязанного к прерыванию. Отвязывание выполняет поток исполнения, привязанный к прерыванию, вызовом функции `KnThreadDetachIrq()`, объявленной в заголовочном файле `sysroot-*-kos/include/coresrv/thread/thread_api.h` из состава KasperskyOS SDK.

Обработка прерывания

После выполнения привязки к прерыванию поток исполнения вызывает функцию `Call()`, объявленную в заголовочном файле `sysroot-*-kos/include/coresrv/syscalls.h` из состава KasperskyOS SDK. В результате этого вызова поток исполнения блокируется. При возникновении прерывания или вызове функции `KnIoDetachIrq()` ядро KasperskyOS отправляет IPC-сообщение процессу, содержащему этот поток. Это IPC-сообщение содержит запрос на обработку прерывания или запрос на отвязывание потока исполнения от прерывания. Когда процесс получает IPC-сообщение, функция `Call()` в потоке исполнения, привязанном к прерыванию, возвращает управление и предоставляет потоку содержимое IPC-сообщения. Поток исполнения извлекает из IPC-сообщения запрос и обрабатывает прерывание либо выполняет отвязывание от прерывания. Если выполняется обработка прерывания, то по ее завершении сведения об успехе или неуспехе обработки добавляются в ответное IPC-сообщение, которое отправляется ядру следующим вызовом функции `Call()` в цикле.

При обработке прерывания нужно использовать функции `IoGetIrqRequest()` и `IoSetIrqAnswer()`, которые объявлены в заголовочном файле `sysroot-*-kos/include/io/io_irq.h` из состава KasperskyOS SDK. Эти функции позволяют извлекать из IPC-сообщений и добавлять в IPC-сообщения данные для информационного обмена между ядром и потоком исполнения, привязанным к прерыванию.

Типовой цикл обработки прерывания включает следующие шаги:

1. Добавление в IPC-сообщение, которое будет отправлено ядру, сведений об успехе или неуспехе обработки прерывания вызовом функции `IoSetIrqAnswer()`.

2. Отправка IPC-сообщения ядру и получение IPC-сообщения от ядра.

   Чтобы выполнить этот шаг, нужно вызвать функции `Call()`. В параметре `handle` требуется указать дескриптор, полученный при вызове функции `KnIoAttachIrq()`. Через параметр `msgOut` необходимо задать IPC-сообщение, которое будет отправлено ядру, а через параметр `msgIn` необходимо задать IPC-сообщение, которое будет получено от ядра.

3. Извлечение запроса из IPC-сообщения, полученного от ядра, вызовом функции `IoGetIrqRequest()`.

4. Обработка прерывания или отвязывание от прерывания в зависимости от запроса.

   Если запрос требует выполнить отвязывание от прерывания, то нужно выйти из цикла обработки прерывания и вызвать функцию `KnThreadDetachIrq()`.

Дерегистрация прерывания

Чтобы дерегистрировать прерывание, нужно выполнить следующие шаги:

1. Выполнить отвязывание потока исполнения от прерывания.

   Чтобы выполнить этот шаг, нужно вызвать функцию `KnThreadDetachIrq()`.

2. Закрыть дескриптор, полученный при вызове функции `KnIoAttachIrq()`.

   Чтобы выполнить этот шаг, нужно вызвать функцию `KnHandleClose()`. (Функция `KnHandleClose()` объявлена в заголовочном файле `sysroot-*-kos/include/coresrv/handle/handle_api.h` из состава KasperskyOS SDK.)

3. Закрыть или отозвать каждый дескриптор прерывания во всех процессах, которые владеют этими дескрипторами.

   Чтобы выполнить этот шаг, нужно использовать функцию `KnHandleClose()` и/или `KnHandleRevoke()`, которые объявлены в заголовочном файле `sysroot-*-kos/include/coresrv/handle/handle_api.h` из состава KasperskyOS SDK.

Одно прерывание может быть зарегистрировано несколько раз, а выполнение этих шагов аннулирует только одну регистрацию. Оставшиеся регистрации продолжают действовать. Каждую регистрацию одного прерывания нужно аннулировать отдельно.

Сведения о функциях API

Функции irq.h

Функция	Сведения о функции
`KnRegisterIrq()`	Назначение Регистрирует прерывание. Параметры • [in] `irq` – номер прерывания. • [out] `outRid` – указатель на дескриптор прерывания. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.

`KnIoAttachIrq()`	Назначение Привязывает вызывающий поток исполнения к прерыванию. Параметры • [in] `rid` – дескриптор прерывания. • [in] `flags` – флаги, задающие параметры прерывания. Флаги определены в заголовочных файлах `sysroot-*-kos/include/io/io_irq.h` и `sysroot-*-kos/include/hal/irqmode.h` из состава KasperskyOS SDK. • [out] `handle` – указатель на клиентский IPC-дескриптор, который используется обработчиком прерывания. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки. Дополнительные сведения В параметре `flags` можно указать следующие флаги: • `IRQ_LEVEL_LOW` – прерывание возникает при низком уровне сигнала. • `IRQ_LEVEL_HIGH` – прерывание возникает при высоком уровне сигнала. • `IRQ_EDGE_RAISE` – прерывание возникает при повышении уровня сигнала. • `IRQ_EDGE_FALL` – прерывание возникает при снижении уровня сигнала. • `IRQ_PRIO_LOW` – прерывание имеет низкий приоритет. • `IRQ_PRIO_NORMAL` – прерывание имеет средний приоритет. • `IRQ_PRIO_HIGH` – прерывание имеет высокий приоритет. • `IRQ_PRIO_RT` – прерывание имеет наивысший приоритет.
`KnIoDetachIrq()`	Назначение Отправляет потоку исполнения запрос, в результате выполнения которого поток должен выполнить отвязывание от прерывания. Параметры • [in] `rid` – дескриптор прерывания. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.
`KnIoEnableIrq()`	Назначение Разрешает (демаскирует) прерывание.

	Параметры • [in] `rid` – дескриптор прерывания. Возвращаемые значения В случае успеха возвращает `rc0k`, иначе возвращает код ошибки.
`KnIoDisableIrq()`	Назначение Запрещает (маскирует) прерывание. Параметры • [in] `rid` – дескриптор прерывания. Возвращаемые значения В случае успеха возвращает `rc0k`, иначе возвращает код ошибки.

Инициализация IPC-транспорта для межпроцессного взаимодействия и управление обработкой IPC-запросов (transport-kos.h, transport-kos-dispatch.h)

API определены в заголовочных файлах `transport-kos.h` и `transport-kos-dispatch.h` из состава KasperskyOS SDK, которые расположены по пути `sysroot-*-kos/include/coresrv/nk`.

Возможности API:

- Инициализировать IPC-транспорт на стороне клиента и на стороне сервера.

  IPC-транспорт – это надстройка над системными вызовами отправки и приема IPC-сообщений, которая позволяет отдельно работать с фиксированной частью и ареной IPC-сообщений. Поверх этой надстройки работает транспортный код.

- Запускать цикл обработки IPC-запросов на стороне сервера.

- Копировать данные в арену IPC-сообщений.

Сведения о функциях API приведены в таблицах ниже.

В этом разделе приведены примеры использования API. В этих примерах программы, которые являются серверами, имеют следующую формальную спецификацию:

`FsDriver.edl`

```
entity 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);
}
```

Инициализация IPC-транспорта для межпроцессного взаимодействия

Чтобы инициализировать IPC-транспорт для взаимодействия с другими процессами, нужно вызвать функцию `NkKosTransport_Init()` или `NkKosTransportSync_Init()`, объявленные в заголовочном файле `transport-kos.h`.

Пример использования функции `NkKosTransport_Init()` на стороне клиента:

```
int main(int argc, const char *argv[])
{
    /* Объявить структуру с параметрами IPC-транспорта */
    NkKosTransport driver_transport;
    /* Объявить прокси-объект. (Тип прокси-объекта является автоматически
     * сгенерированным транспортным кодом.) */
    struct FileIface_proxy file_operations_proxy;
    /* Объявить структуры для сохранения фиксированной части IPC-запроса и
     * IPC-ответа для метода службы. (Типы структур являются автоматически
     * сгенерированным транспортным кодом.) */
    struct FileIface_Open_req req;
    struct FileIface_Open_res res;
    /* Получить клиентский IPC-дескриптор и идентификатор службы */
    Handle driver_handle;
    rtl_uint32_t file_operations_riid;
    if (KnCmConnect("FsDriver", "operationsComp.fileOperations", INFINITE_TIMEOUT,
                &driver_handle, &file_operations_riid) == rcOk) {
        /* Инициализировать структуру с параметрами IPC-транспорта */
        NkKosTransport_Init(&driver_transport, driver_handle, NK_NULL, 0);
        /* Инициализировать прокси-объект. (Метод инициализации прокси-объекта
         * является автоматически сгенерированным транспортным кодом.) */
        FileIface_proxy_init(&file_operations_proxy, &driver_transport.base,
                             (nk_iid_t) file_operations_riid);
    }
...
    /* Вызвать метод службы. (Метод является автоматически
     * сгенерированным транспортным кодом.) */
    strncpy(req.path, "/example/file/path", sizeof(req.path));
    if (FileIface_Open(file_operations_proxy.base, &req, NULL,
                       &res, NULL) != NK_EOK) {
...
    }
...
}
```

Если клиенту требуется использовать несколько служб, то нужно инициализировать столько же прокси-объектов. При инициализации каждого прокси-объекта нужно указать IPC-транспорт, ассоциированный через клиентский IPC-дескриптор с нужным сервером. При инициализации нескольких прокси-объектов, относящихся к службам одного сервера, можно указать один и тот же IPC-транспорт, который ассоциирован с этим сервером.

Пример использования функции `NkKosTransport_Init()` на стороне сервера:

```
int main(int argc, const char *argv[])
{
...
    /* Объявить структуру с параметрами IPC-транспорта */
    NkKosTransport transport;
    /* Получить слушающий дескриптор. (Идентификатор службы
     * FsDriver_operationsComp_fileOperations_iid является
     * автоматически сгенерированным транспортным кодом.) */
    Handle handle;
    char client[32];
    char endpoint[32];
    Retcode rc = KnCmListen(RTL_NULL, INFINITE_TIMEOUT, client, endpoint);
    if (rc == rcOk)
        rc = KnCmAccept(client, endpoint,
                        FsDriver_operationsComp_fileOperations_iid,
                        INVALID_HANDLE, &handle);
...
    /* Инициализировать структуру с параметрами IPC-транспорта */
    NkKosTransport_Init(&transport, handle, NK_NULL, 0);
...
    /* Цикл обработки IPC-запросов */
    do
    {
...
        /* Получить IPC-запрос */
        rc = nk_transport_recv(&transport.base, ...);
        if (rc == NK_EOK) {
            /* Обработать IPC-запрос вызовом диспетчера. (Диспетчер
             * является автоматически сгенерированным транспортным
             * кодом.) */
            rc = FsDriver_entity_dispatch(...);
            if (rc == NK_EOK) {
                /* Отправить IPC-ответ */
                rc = nk_transport_reply(&transport.base, ...);
            }
        }
    }
    while (rc == NK_EOK)
    return EXIT_SUCCESS;
}
```

Если сервер обрабатывает IPC-запросы, поступающие через несколько IPC-каналов, то нужно учитывать следующие особенности:

- Если слушающий дескриптор ассоциирован со всеми IPC-каналами, то для IPC-взаимодействия со всеми клиентами можно использовать один IPC-транспорт, ассоциированный с этим слушающим дескриптором.

- Если IPC-каналы ассоциированы с разными слушающими дескрипторами, то для IPC-взаимодействия с каждой группой клиентов, соответствующих одному слушающему дескриптору, нужно использовать отдельный IPC-транспорт, ассоциированный с этим слушающим дескриптором. При этом обработку IPC-

запросов можно выполнять в параллельных потоках исполнения, если реализация методов служб потокобезопасна.

Функция `NkKosTransportSync_Init()` позволяет инициализировать IPC-транспорт с поддержкой прерывания блокирующих системных вызовов `Call()` и `Recv()`. (Прерывание этих вызовов может потребоваться, например, чтобы корректно завершить процесс, который их выполняет.) Чтобы прерывать системные вызовы `Call()` и `Recv()`, нужно использовать API ipc_api.h.

Функция `NkKosSetTransportTimeouts()`, объявленная в заголовочном файле `transport-kos.h`, позволяет задать для IPC-транспорта максимальное время блокировки системных вызовов `Call()` и `Recv()`.

Запуск цикла обработки IPC-запросов

Цикл обработки IPC-запроса на сервере включает следующие стадии:

1. Получение IPC-запроса.
2. Обработка IPC-запроса.
3. Отправка IPC-ответа.

Каждую стадию этого цикла можно выполнять отдельно, последовательно вызывая функции `nk_transport_recv()`, диспетчер, `nk_transport_reply()`. (Функции `nk_transport_recv()` и `nk_transport_reply()` объявлены в заголовочном файле `sysroot-*-kos/include/nk/transport.h` из состава KasperskyOS SDK.) А можно вызвать функцию `NkKosTransport_Dispatch()` или `NkKosDoDispatch()`, внутри которых этот цикл выполняется полностью. (Функции `NkKosTransport_Dispatch()` и `NkKosDoDispatch()` объявлены в заголовочных файлах `transport-kos.h` и `transport-kos-dispatch.h` соответственно.) Использовать функцию `NkKosDoDispatch()` удобнее, поскольку нужно выполнять меньше подготовительных операций (в том числе не требуется инициализировать IPC-транспорт).

Для инициализации структуры, передаваемой функции `NkKosDoDispatch()` через параметр `info`, можно использовать макросы, определенные в заголовочном файле `transport-kos-dispatch.h`.

Функции `NkKosTransport_Dispatch()` и `NkKosDoDispatch()` можно вызывать из параллельных потоков исполнения, если реализация методов служб потокобезопасна.

Пример использования функции `NkKosDoDispatch()`:

```
/* Функция реализует метод службы. */
static nk_err_t Open_impl(...)
{
...
}

/* Функция реализует метод службы. */
static nk_err_t Read_impl(...)
{
...
}

/* Функция инициализирует указатели на функции, реализующие методы службы.
 * (Эти указатели используются диспетчером для вызова функций, реализующих
 * методы службы. Типы структур являются автоматически сгенерированным
 * транспортным кодом.) */
```

```
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[])
{
...
    /* Объявить структуру, которая требуется функции
     * NkKosDoDispatch() для использования транспортного кода. */
    NkKosDispatchInfo info;
    /* Объявить стабы. (Типы стабов являются автоматически сгенерированным
     * транспортным кодом. */
    struct Operations_component component;
    struct FsDriver_entity entity;
    /* Получить слушающий дескриптор */
    Handle handle = ServiceLocatorRegister("driver_connection", NULL, 0, &iid);
    assert(handle != INVALID_HANDLE);
    /* Инициализировать стабы. (Методы инициализации стабов являются
     * автоматически сгенерированным транспортным кодом. Функция
     * CreateFileOperations() реализована разработчиком решения на
     * базе KasperskyOS, чтобы инициализировать
     * указатели на функции, реализующие методы службы.) */
    Operations_component_init(&component, CreateFileOperations());
    FsDriver_entity_init(&entity, &component);
    /* Инициализировать структуру, которая требуется функции
     * NkKosDoDispatch() для использования транспортного кода. */
    info = NK_TASK_DISPATCH_INFO_INITIALIZER(FsDriver, entity);
    /* Запустить цикл обработки IPC-запросов */
    NkKosDoDispatch(handle, info);
    return EXIT_SUCCESS;
}
```

Пример использования функции `NkKosTransport_Dispatch()`:

```
/* Функция реализует метод службы. */
static nk_err_t Open_impl(...)
{
...
}

/* Функция реализует метод службы. */
static nk_err_t Read_impl(...)
{
...
}

/* Функция инициализирует указатели на функции, реализующие методы службы.
 * (Эти указатели используются диспетчером для вызова функций, реализующих
 * методы службы. Типы структур являются автоматически сгенерированным
 * транспортным кодом.) */
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[])
{
...
    /* Объявить структуру с параметрами IPC-транспорта */
    NkKosTransport transport;
    /* Объявить стабы. (Типы стабов являются автоматически сгенерированным
     * транспортным кодом. */
    struct Operations_component component;
    struct FsDriver_entity entity;
    /* Объявить объединения фиксированной части IPC-запросов и
     * IPC-ответов. (Типы объединений являются автоматически сгенерированным
     * транспортным кодом.) */
    union FsDriver_entity_req req;
    union FsDriver_entity_res res;
    /* Объявить массив для арены IPC-ответов. (Размер массива является
     * автоматически сгенерированным транспортным кодом.) */
    char res_buffer[FsDriver_entity_res_arena_size];
    /* Объявить и инициализировать дескриптор арены IPC-ответов.
     * (Тип дескриптора и макрос его инициализации определены в заголовочном файле
     * sysroot-*-kos/include/nk/arena.h из состава KasperskyOS SDK.) */
    struct nk_arena res_arena = NK_ARENA_INITIALIZER(res_buffer,
                                                     res_buffer + sizeof(res_buffer));
    /* Получить слушающий дескриптор */
    Handle handle = ServiceLocatorRegister("driver_connection", NULL, 0, &iid);
    assert(handle != INVALID_HANDLE);
    /* Инициализировать структуру с параметрами IPC-транспорта */
    NkKosTransport_Init(&transport, handle, NK_NULL, 0);
    /* Инициализировать стабы. (Методы инициализации стабов являются
     * автоматически сгенерированным транспортным кодом. Функция
     * CreateFileOperations() реализована разработчиком решения на
     * базе KasperskyOS, чтобы инициализировать
     * указатели на функции, реализующие методы службы.) */
    Operations_component_init(&component, CreateFileOperations());
    FsDriver_entity_init(&entity, &component);
    /* Запустить цикл обработки IPC-запросов. (Диспетчер FsDriver_entity_dispatch
     * является автоматически сгенерированным транспортным кодом.) */
    NkKosTransport_Dispatch(&transport.base, FsDriver_entity_dispatch,
                            &entity, &req, sizeof(FsDriver_entity_req),
                            RTL_NULL, &res, &res_arena);
    return EXIT_SUCCESS;
}
```

Копирование данных в арену IPC-сообщений

Чтобы скопировать строку в арену IPC-сообщений, нужно вызвать функцию `NkKosCopyStringToArena()`, объявленную в заголовочном файле `transport-kos.h`. Эта функция резервирует участок арены и копирует строку в этот участок.

Пример использования функции `NkKosCopyStringToArena()`:

```
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)
{
    /* Скопировать строку в арену IPC-ответов */
    if (NkKosCopyStringToArena(&res_arena, &res.content,
                               "CONTENT OF THE FILE") != rcOk) {
...
    }
    return NK_EOK;
}
```

Сведения о функциях API

Функции transport-kos.h

Функция	Сведения о функции
`NkKosTransport_Init()`	Назначение Инициализирует IPC-транспорт. Параметры • [out] `transport` – указатель на структуру с параметрами IPC-транспорта. • [in] `handle` – клиентский или серверный IPC-дескриптор. • [in] `view` – параметр, который должен иметь значение `NK_NULL`. • [in] `size` – параметр, который должен иметь значение `0`. Возвращаемые значения Нет.
`NkKosTransportSync_Init()`	Назначение Инициализирует IPC-транспорт с поддержкой прерывания системных вызовов `Call()` и/или `Recv()`. Параметры • [out] `transport` – указатель на структуру с параметрами IPC-транспорта. • [in] `handle` – клиентский или серверный IPC-дескриптор. • [in,optional] `callSyncHandle` – дескриптор объекта синхронизации IPC для системных вызовов `Call()` или `INVALID_HANDLE`, если прерывание системных вызовов `Call()` не требуется.

	• [in,optional] `recvSyncHandle` – дескриптор объекта синхронизации IPC для системных вызовов `Recv()` или `INVALID_HANDLE`, если прерывание системных вызовов `Recv()` не требуется. Возвращаемые значения Нет.
`NkKosSetTransportTimeouts()`	Назначение Задает для IPC-транспорта максимальное время блокировки системных вызовов `Call()` и `Recv()`. Параметры • [out] `transport` – указатель на структуру с параметрами IPC-транспорта. • [in] `recvTimeout` – максимальное время блокировки системных вызовов `Recv()` в миллисекундах или `INFINITE_TIMEOUT`, чтобы задать неограниченное время блокировки. • [in] `callTimeout` – максимальное время блокировки системных вызовов `Call()` в миллисекундах или `INFINITE_TIMEOUT`, чтобы задать неограниченное время блокировки. Возвращаемые значения Нет.
`NkKosTransport_Dispatch()`	Назначение Запускает цикл обработки IPC-запросов. Параметры • [in] `transport` – указатель на поле `base` структуры с параметрами IPC-транспорта. • [in] `dispatch` – указатель на диспетчер (dispatch-метод) из транспортного кода. Диспетчер имеет имя `<имя класса процессов>_entity_dispatch`. • [in] `impl` – указатель на стаб, который представляет собой структуру с типом `<имя класса процессов>_entity` из транспортного кода. Через эту структуру диспетчер получает указатели на функции, реализующие методы служб. • [out] `req` – указатель на объединение с типом `<имя класса процессов>_entity_req` из транспортного кода. Это объединение предназначено для сохранения фиксированной части IPC-запросов для любых методов служб, предоставляемых сервером. • [in] `req_size` – максимальный размер фиксированной части IPC-запросов в байтах. Определяется как `sizeof(<имя класса процессов>_entity_req)`, где

	`<имя класса процессов>_entity_req` является типом из транспортного кода. • [in,out,optional] `req_arena` – указатель на дескриптор арены IPC-запросов или `RTL_NULL`, если арена IPC-запросов не используется. Тип дескриптора определен в заголовочном файле `sysroot-*-kos/include/nk/arena.h` из состава KasperskyOS SDK. • [out] `res` – указатель на объединение с типом `<имя класса процессов>_entity_res` из транспортного кода. Это объединение предназначено для сохранения фиксированной части IPC-ответов для любых методов служб, предоставляемых сервером. • [in,out,optional] `res_arena` – указатель на дескриптор арены IPC-ответов или `RTL_NULL`, если арена IPC-ответов не используется. Тип дескриптора определен в заголовочном файле `sysroot-*-kos/include/nk/arena.h` из состава KasperskyOS SDK. Возвращаемые значения В случае неуспеха возвращает код ошибки.
`NkKosCopyStringToArena()`	Назначение Резервирует участок арены и копирует строку в этот участок. Параметры • [in,out] `arena` – указатель на дескриптор арены. Тип дескриптора определен в заголовочном файле `sysroot-*-kos/include/nk/arena.h` из состава KasperskyOS SDK. • [out] `field` – указатель на дескриптор участка арены, куда скопирована строка. Тип дескриптора определен в заголовочном файле `sysroot-*-kos/include/nk/types.h`. • [in] `src` – указатель на строку для копирования в арену IPC-сообщений. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.

Функции transport-kos-dispatch.h

Функция	Сведения о функции
`NkKosDoDispatch()`	Назначение Запускает цикл обработки IPC-запросов. Параметры • [in] `h` – серверный IPC-дескриптор.

- [in] `info` – указатель на структуру, содержащую данные, которые требуются функции для использования транспортного кода (включая имена типов, размеры фиксированной части и арены IPC-сообщений).

Возвращаемые значения

Нет.

Инициализация IPC-транспорта для обращения к модулю безопасности (transport-kos-security.h)

API определен в заголовочном файле `sysroot-*-kos/include/coresrv/nk/transport-kos-security.h` из состава KasperskyOS SDK.

API позволяет инициализировать IPC-транспорт для обращения к модулю безопасности Kaspersky Security Module через интерфейс безопасности. Поверх IPC-транспорта работает транспортный код.

Сведения о функциях API приведены в таблице ниже.

В этом разделе приведен пример использования API. В этом примере программа, которая обращается к модулю безопасности, имеет следующую формальную спецификацию:

Verifier.edl

```
entity Verifier
security Approve
```

Approve.idl

```
package Approve
interface {
    Check(in UInt32 port);
}
```

Фрагмент описания политики в примере:

security.psl

```
...
security src=Verifier, method=Check { assert (message.port > 80) }
...
```

Использование API

Чтобы инициализировать IPC-транспорт для обращения к модулю безопасности, нужно вызвать функцию `NkKosSecurityTransport_Init()`.

Пример использования функции `NkKosSecurityTransport_Init()`:

```c
int main(void)
{
    /* Объявить структуру с параметрами IPC-транспорта для обращения к
     * модулю безопасности */
    NkKosSecurityTransport security_transport;
    /* Объявить прокси-объект. (Тип прокси-объекта является автоматически
     * сгенерированным транспортным кодом.) */
    struct Approve_proxy security_proxy;
    /* Объявить структуры для сохранения фиксированной части IPC-запроса и IPC-ответа
для
     * метода интерфейса безопасности. (Типы структур являются автоматически
сгенерированным
     * транспортным кодом.) */
    struct Approve_Check_req security_req;
    struct Approve_Check_res security_res;
    /* Инициализировать структуру с параметрами IPC-транспорта для обращения к
     * модулю безопасности */
    if (NkKosSecurityTransport_Init(&security_transport, NK_NULL, 0) == NK_EOK) {
        /* Инициализировать прокси-объект. (Метод инициализации прокси-объекта и
         * идентификатор интерфейса безопасности Verifier_securityIid
         * являются автоматически сгенерированным транспортным кодом.) */
        Approve_proxy_init(&security_proxy, &security_transport.base,
Verifier_securityIid);
    }
...
    /* Вызвать метод интерфейса безопасности. (Метод является автоматически
сгенерированным
     * транспортным кодом. Метод не передает через параметр security_res никакие
данные.
     * Указать этот параметр нужно только потому, что этого требует реализация
метода.) */
    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;
}
```

Если процессу требуется использовать несколько интерфейсов безопасности, то нужно инициализировать столько же прокси-объектов, указав один и тот же IPC-транспорт и уникальные идентификаторы интерфейсов безопасности.

Сведения о функциях API

Функции transport-kos-security.h

Функция	Сведения о функции
NkKosSecurityTransport_Init()	Назначение Инициализирует IPC-транспорт для обращения к модулю безопасности Kaspersky Security Module через интерфейс безопасности.

	Параметры • [out] `transport` – указатель на структуру с параметрами IPC-транспорта для обращения к модулю безопасности. • [in] `view` – параметр, который должен иметь значение `NK_NULL`. • [in] `size` – параметр, который должен иметь значение `0`. Возвращаемые значения В случае успеха возвращает `NK_EOK`, иначе возвращает код ошибки.

Генерация случайных чисел (random_api.h)

API определен в заголовочном файле `sysroot-*-kos/include/kos/random/random_api.h` из состава KasperskyOS SDK.

API позволяет генерировать случайные числа, а также включает функции, которые можно использовать, чтобы обеспечить высокую энтропию (высокий уровень непредсказуемости) начального значения генератора случайных чисел. *Начальное значение генератора случайных чисел* (англ. seed) определяет последовательность генерируемых случайных чисел. То есть после установки одного и того же начального значения генератор создает одинаковые последовательности случайных чисел. (Энтропия таких чисел полностью определяется энтропией начального значения, поэтому точнее их называть не случайными, а псевдослучайными.)

Генератор случайных чисел одного процесса не зависит от генераторов случайных чисел других процессов.

Сведения о функциях API приведены в таблице ниже.

Использование API

Чтобы сгенерировать последовательность случайных байтовых значений, нужно вызвать функцию `KosRandomGenerate()` или `KosRandomGenerateEx()`.

Пример использования функции `KosRandomGenerate()`:

```
size_t random_number;
if (KosRandomGenerate(sizeof random_number, &random_number) == rcOk) {
...
}
```

Функция `KosRandomGenerateEx()` позволяет получить уровень качества сгенерированных случайных значений через выходной параметр `quality`. Уровень качества может быть высоким или низким. Высококачественными считаются случайные значения, которые сгенерированы при выполнении всех следующих условий:

1. На момент изменения начального значения генератора случайных чисел зарегистрирован хотя бы один источник энтропии, и успешно получены данные из всех зарегистрированных источников энтропии.

Чтобы зарегистрировать источник энтропии, нужно вызвать функцию `KosRandomRegisterSrc()`. Через параметр `callback` эта функция принимает указатель на callback-функцию следующего типа:

```
typedef Retcode (*KosRandomSeedMethod)(void      *context,
                                       rtl_size_t size,
                                       void      *output);
```

Эта callback-функция должна, используя параметры `context`, получить данные размером `size` байт из источника энтропии и записать их в буфер `output`. Если функция возвращает `rcOk`, то считается, что данные из источника энтропии были получены успешно. Источником энтропии может быть, например, оцифрованный сигнал какого-либо датчика или аппаратный генератор случайных чисел.

Чтобы дерегистрировать источник энтропии, нужно вызвать функцию `KosRandomUnregisterSrc()`.

2. Выполнена инициализация генератора случайных чисел, если до изменения начального значения генератора случайных чисел с выполнением условия 1 уровень качества был низким.

   Если уровень качества низкий, то сделать его высоким без инициализации генератора случайных чисел нельзя.

   Чтобы инициализировать генератор случайных чисел, нужно вызвать функцию `KosRandomInitSeed()`. Энтропию данных, передаваемых через параметр `seed`, должен гарантировать вызывающий процесс.

3. Счетчик случайных байтовых значений, которые сгенерированы после изменения начального значения генератора случайных чисел с выполнением условия 1, не превышает заданный в системе лимит.

4. Счетчик времени, которое прошло после изменения начального значения генератора случайных чисел с выполнением условия 1, не превышает заданный в системе лимит.

Если хотя бы одно из этих условий не выполняется, сгенерированные случайные значения считаются низкокачественными.

При запуске процесса начальное значение генератора случайных чисел задается системой автоматически. Затем начальное значение генератора случайных чисел изменяется в следующих случаях:

- Генерация последовательности случайных значений.

  Каждый успешный вызов функции `KosRandomGenerateEx()` с параметром `size` больше нуля и каждый успешный вызов функции `KosRandomGenerate()` изменяют начальное значение генератора случайных чисел, но не каждое такое изменение выполняется с получением данных из зарегистрированных источников энтропии. Зарегистрированные источники энтропии используются только при нарушении условия 3 или 4. При этом, если зарегистрирован хотя бы один источник энтропии, то сбрасывается счетчик времени изменения начального значения генератора случайных чисел. А если данные из хотя бы одного источника энтропии получены успешно, то также сбрасывается счетчик сгенерированных случайных байтовых значений.

  Уровень качества может измениться с высокого на низкий.

- Инициализация генератора случайных чисел.

  Каждый успешный вызов функции `KosRandomInitSeed()` изменяет начальное значение генератора случайных чисел, используя данные, переданные через параметр `seed` и полученные из зарегистрированных источников энтропии. При наличии хотя бы одного зарегистрированного источника энтропии сбрасываются счетчики сгенерированных случайных байтовых значений и времени изменения начального значения генератора случайных чисел. В противном случае сбрасывается только счетчик сгенерированных случайных байтовых значений.

  Уровень качества может измениться с высокого на низкий и наоборот.

- Регистрация источника энтропии.

Каждый успешный вызов функции `KosRandomRegisterSrc()` изменяет начальное значение генератора случайных чисел, используя данные только из регистрируемого источника энтропии. При этом сбрасывается счетчик сгенерированных случайных байтовых значений.

Уровень качества не может измениться.

Возможный сценарий генерации случайных значений высокого качества включает следующие шаги:

1. Зарегистрировать хотя бы один источник энтропии вызовом функции `KosRandomRegisterSrc()`.
2. Сгенерировать последовательность случайных значений вызовом функции `KosRandomGenerateEx()`.
3. Проверить уровень качества случайных значений.

   Если уровень качества низкий, инициализировать генератор случайных чисел вызовом функции `KosRandomInitSeed()` и перейти к шагу 2.

   Если уровень качестве высокий, перейти к шагу 4.
4. Использовать случайные значения.

Чтобы получить уровень качества без генерации случайных значений, нужно вызвать функцию `KosRandomGenerateEx()` со значениями `0` и `RTL_NULL` в параметрах `size` и `output` соответственно.

Сведения о функциях API

Функции random_api.h

Функция	Сведения о функции
`KosRandomInitSeed()`	Назначение Инициализирует генератор случайных чисел. Параметры • [in] `seed` – указатель на байтовый массив, который используется, чтобы изменить начальное значение генератора случайных чисел. Массив должен иметь размер 32 байта. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.
`KosRandomGenerate()`	Назначение Генерирует последовательность случайных байтовых значений. Параметры • [in] `size` – размер буфера для сохранения последовательности (в байтах). • [out] `output` – указатель на буфер для сохранения последовательности. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.

`KosRandomGenerateEx()`	Назначение Генерирует последовательность случайных байтовых значений. Параметры • [in] `size` – размер буфера для сохранения последовательности (в байтах). • [out] `output` – указатель на буфер для сохранения последовательности. • [out] `quality` – указатель на булевое значение, которое истинно, если сгенерированные случайные значения имеют высокое качество, и ложно, если сгенерированные случайные значения имеют низкое качество. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.
`KosRandomRegisterSrc()`	Назначение Регистрирует источник энтропии. Параметры • [in] `callback` – указатель на функцию, которая позволяет получить данные из источника энтропии для изменения начального значения генератора случайных чисел. • [in,optional] `context` – указатель на параметры, передаваемые функции, заданной через параметр `callback`, или `RTL_NULL`, если параметров нет. • [in] `size` – размер данных (в байтах), которые нужно получить из источника энтропии при вызове функции, заданной через параметр `callback`. Должен быть не менее 32 байт. • [out] `handle` – адрес указателя, используемого для дерегистрации источника энтропии. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.
`KosRandomUnregisterSrc()`	Назначение Дерегистрирует источник энтропии. Параметры • [in] `handle` – указатель, который получен при регистрации источника энтропии. Возвращаемые значения

	В случае успеха возвращает `rc0k`, иначе возвращает код ошибки.

Получение и изменение значений времени (time_api.h)

API определен в заголовочном файле `sysroot-*-kos/include/coresrv/time/time_api.h` из состава KasperskyOS SDK.

Основные возможности API:

- получать и изменять системное время;
- получать монотонное время, отсчитанное с момента запуска ядра KasperskyOS;
- получать разрешение источников системного и монотонного времени.

Сведения о функциях API приведены в таблице ниже.

Функции time_api.h

Функция	Сведения о функции
`KnGetSystemTimeRes()`	Назначение Позволяет получить разрешение источника системного времени. Параметры • [out] `res` – указатель на структуру, содержащую в поле `nsec` разрешение источника системного времени в наносекундах. Тип структуры определен в заголовочном файле `sysroot-*-kos/include/rtl/rtc.h` из состава KasperskyOS SDK. Возвращаемые значения В случае успеха возвращает `rc0k`, иначе возвращает код ошибки.
`KnSetSystemTime()`	Назначение Задает системное время. Параметры • [in] `time` – указатель на структуру, содержащую следующие сведения: в поле `sec` – число секунд, прошедших с 1 января 1970 года; в поле `nsec` – число наносекунд, прошедших с момента, заданного в поле `sec`. Тип структуры определен в заголовочном файле `sysroot-*-kos/include/rtl/rtc.h` из состава KasperskyOS SDK. Возвращаемые значения В случае успеха возвращает `rc0k`, иначе возвращает код ошибки.
`KnGetSystemTime()`	Назначение Позволяет получить системное время.

	Параметры • [out] `time` – указатель на структуру, содержащую следующие сведения: в поле `sec` – число секунд, прошедших с 1 января 1970 года; в поле `nsec` – число наносекунд, прошедших с момента, заданного в поле `sec`. Тип структуры определен в заголовочном файле `sysroot-*-kos/include/rtl/rtc.h` из состава KasperskyOS SDK. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.
`KnGetUpTimeRes()`	Назначение Позволяет получить разрешение источника монотонного времени, отсчитываемого с момента запуска ядра KasperskyOS. Параметры • [out] `res` – указатель на структуру, содержащую в поле `nsec` разрешение источника монотонного времени, отсчитываемого с момента запуска ядра KasperskyOS, в наносекундах. Тип структуры определен в заголовочном файле `sysroot-*-kos/include/rtl/rtc.h` из состава KasperskyOS SDK. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.
`KnGetUpTime()`	Назначение Позволяет получить монотонное время, отсчитанное с момента запуска ядра KasperskyOS. Параметры • [out] `time` – указатель на структуру, содержащую следующие сведения: в поле `sec` – число секунд, прошедших с момента запуска ядра KasperskyOS; в поле `nsec` – число наносекунд, прошедших с момента, заданного в поле `sec`. Тип структуры определен в заголовочном файле `sysroot-*-kos/include/rtl/rtc.h` из состава KasperskyOS SDK. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.
`KnGetRtcTime()`	Назначение Позволяет получить системное время. Параметры • [out] `rt` – указатель на структуру, содержащую следующие сведения о времени: год, месяц, день, часы, минуты, секунды, миллисекунды. Тип структуры определен в заголовочном файле `sysroot-*-kos/include/rtl/rtc.h` из состава KasperskyOS SDK.

	Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.
`KnGetMSecSinceStart()`	Назначение Позволяет получить монотонное время, отсчитанное с момента запуска ядра KasperskyOS. Параметры Нет. Возвращаемые значения Монотонное время, отсчитанное с момента запуска ядра KasperskyOS, в миллисекундах.
`KnAdjSystemTime()`	Назначение Запускает постепенную корректировку системного времени. Параметры • [in] `adj` – указатель на структуру, содержащую интервал времени, на который нужно скорректировать системное время (*sec*10^9+nsec* наносекунд) или `RTL_NULL`, если не требуется запускать корректировку, а нужно только получить сведения о ранее запущенной корректировке (через параметр `prev`). Тип структуры определен в заголовочном файле `sysroot-*-kos/include/rtl/rtc.h` из состава KasperskyOS SDK. • [in] `slew` – скорость корректировки системного времени (микросекунд в секунду). • [out] `prev` – указатель на структуру, содержащую интервал времени, отражающий, на какое значение оставалось (или остается в случае указания `RTL_NULL` в параметре `adj`) скорректировать системное время, чтобы уже запущенная постепенная корректировка была полностью завершена (*sec*10^9+nsec* наносекунд). Тип структуры определен в заголовочном файле `sysroot-*-kos/include/rtl/rtc.h` из состава KasperskyOS SDK. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки. Дополнительные сведения Если запустить новую корректировку до завершения ранее запущенной, то ранее запущенная будет прервана.

Использование уведомлений (notice_api.h)

API определен в заголовочном файле `sysroot-*-kos/include/coresrv/handle/notice_api.h` из состава KasperskyOS SDK.

API позволяет отслеживать события, которые происходят с ресурсами (как системными, так и пользовательскими), а также уведомлять о событиях, происходящих с пользовательскими ресурсами, другие процессы и потоки исполнения.

Сведения о функциях API приведены в таблице ниже.

Использование API

Механизм уведомлений использует маску событий. *Маска событий* – значение, биты которого интерпретируются как события, которые должны отслеживаться или уже произошли. Маска событий имеет размер 32 бита и состоит из общей и специальной части. Общая часть описывает события, неспецифичные для любых ресурсов. Специальная часть описывает события, специфичные для ресурсов. Флаги специальной части для системных ресурсов и флаги общей части определены в заголовочном файле `sysroot-*-kos/include/handle/event_descr.h` из состава KasperskyOS SDK. (Например, флаг общей части `EVENT_OBJECT_DESTROYED` означает прекращение существования ресурса, а флаг специальной части `EVENT_TASK_COMPLETED` означает завершение процесса.) Флаги специальной части для пользовательского ресурса определяются поставщиком этого ресурса с использованием макросов `OBJECT_EVENT_SPEC()` и `OBJECT_EVENT_USER()`, которые определены в заголовочном файле `sysroot-*-kos/include/handle/event_descr.h` из состава KasperskyOS SDK. Поставщику ресурса необходимо экспортировать публичные заголовочные файлы с описанием флагов специальной части.

Типовой сценарий получения уведомлений о событиях, происходящих с ресурсами, включает следующие шаги:

1. Создание *приемника уведомлений* (объекта ядра KasperskyOS, в котором накапливаются уведомления) вызовом функции `KnNoticeCreate()`.

2. Добавление в приемник уведомлений записей вида "ресурс – маска событий", чтобы настроить его на получение уведомлений о событиях, которые происходят с интересующими ресурсами.

   Чтобы добавить запись вида "ресурс – маска событий" в приемник уведомлений, нужно вызвать функцию `KnNoticeSubscribeToObject()`. (В маске прав дескриптора ресурса, указанного в параметре `object`, должен быть флаг `OCAP_HANDLE_GET_EVENT`.) Для одного и того же ресурса можно добавить несколько записей вида "ресурс – маска событий", при этом не требуется, чтобы идентификаторы этих записей были уникальными. Отслеживаемые события для каждой записи вида "ресурс – маска событий" нужно задать маской событий, которая может соответствовать одному или нескольким событиям.

   Добавленные в приемник уведомлений записи вида "ресурс – маска событий" можно полностью или частично удалить, чтобы этот приемник не получал уведомления, соответствующие этим записям. Чтобы удалить из приемника уведомлений все записи вида "ресурс – маска событий", нужно вызвать функцию `KnNoticeDropAndWake()`. Чтобы удалить из приемника уведомлений записи вида "ресурс – маска событий", относящиеся к одному ресурсу, нужно вызвать функцию `KnNoticeUnsubscribeFromObject()`. Чтобы удалить из приемника уведомлений записи вида "ресурс – маска событий" с конкретным идентификатором, нужно вызвать функцию `KnNoticeUnsubscribeFromEvent()`.

   Записи вида "ресурс – маска событий" можно добавлять в приемник уведомлений и удалять из него на протяжении всего жизненного цикла этого приемника уведомлений.

3. Извлечение уведомлений из приемника с использованием функции `KnNoticeGetEvent()`.

   При вызове функции `KnNoticeGetEvent()` можно задать время ожидания появления уведомлений в приемнике. Потоки, заблокированные в ожидании появления уведомлений в приемнике, возобновят свое исполнение при появлении уведомлений, даже если эти уведомления соответствуют записям вида "ресурс – маска событий", добавленным после начала ожидания.

Потоки, заблокированные в ожидании появления уведомлений в приемнике, возобновят свое исполнение, если из этого приемника будут удалены все записи вида "ресурс – маска событий" вызовом функции `KnNoticeDropAndWake()`. Если после вызова функции `KnNoticeDropAndWake()` добавить в приемник уведомлений хотя бы одну запись вида "ресурс – маска событий", то потоки исполнения, получающие уведомления из этого приемника, будут снова заблокированы при вызове функции `KnNoticeGetEvent()` на заданное время ожидания при отсутствии уведомлений. Если все записи вида "ресурс – маска событий" удалены из приемника уведомлений с использованием функции `KnNoticeUnsubscribeFromObject()` и/или функции `KnNoticeUnsubscribeFromEvent()`, то исполнение потоков, ожидающих появления уведомлений в этом приемнике, не возобновляется до истечения времени ожидания.

4. Удаление приемника уведомлений вызовом функции `KnNoticeRelease()`.

   Потоки, заблокированные в ожидании появления уведомлений в приемнике, возобновят свое исполнение при удалении этого приемника вызовом функции `KnNoticeRelease()`.

Чтобы уведомить другие процессы и/или потоки исполнения о событиях, которые произошли с пользовательским ресурсом, нужно вызвать функцию `KnNoticeSetObjectEvent()`. В результате вызова этой функции появляются уведомления в приемниках, настроенных на получение уведомлений о событиях, заданных через параметр `evMask`, которые происходят с пользовательским ресурсом, заданным через параметр `object`. В параметре `evMask` нельзя указывать флаги общей части маски событий, так как о событиях, соответствующих общей части маски событий, может сигнализировать только ядро. Если процесс, вызывающий функцию `KnNoticeSetObjectEvent()`, создал дескриптор пользовательского ресурса, указанный в параметре `object`, то в параметре `evMask` можно указать флаги, которые определены макросами `OBJECT_EVENT_SPEC()` и `OBJECT_EVENT_USER()`. Если процесс, вызывающий функцию `KnNoticeSetObjectEvent()`, получил от другого процесса дескриптор пользовательского ресурса, указанный в параметре `object`, то в параметре `evMask` можно указать только те флаги, которые определены макросом `OBJECT_EVENT_USER()`, при этом в маске прав полученного дескриптора должен быть флаг `OCAP_HANDLE_SET_EVENT`.

Сведения о функциях API

Функции notice_api.h

Функция	Сведения о функции
`KnNoticeCreate()`	Назначение Создает приемник уведомлений. Параметры • [out] `notice` – указатель на идентификатор приемника уведомлений. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.
`KnNoticeSubscribeToObject()`	Назначение Добавляет запись вида "ресурс – маска событий" в приемник уведомлений, чтобы он получал уведомления о событиях, которые происходят с заданным ресурсом и соответствуют заданной маске событий. Параметры

	• [in] `notice` – идентификатор приемника уведомлений. • [in] `object` – дескриптор ресурса. • [in] `evMask` – маска событий. • [in] `evId` – идентификатор записи вида "ресурс – маска событий". Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.
`KnNoticeGetEvent()`	Назначение Извлекает уведомления из приемника. Параметры • [in] `notice` – идентификатор приемника уведомлений. • [in] `msec` – время ожидания появления уведомлений в приемнике в миллисекундах или `INFINITE_TIMEOUT`, чтобы задать неограниченное время ожидания. • [in] `countMax` – максимальное число уведомлений, извлекаемое за один вызов функции. • [out] `events` – указатель на набор уведомлений, которые представляют собой структуры, содержащие идентификатор записи вида "ресурс – маска событий" и маску событий, произошедших с ресурсом. • [out] `count` – число извлеченных уведомлений. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки. Если время ожидания появления уведомлений в приемники истекло, возвращает `rcTimeout`. Если ожидание появления уведомлений в приемнике прервано вызовом функции `KnNoticeRelease()` или `KnNoticeDropAndWake()`, возвращает `rcResourceNotFound`.
`KnNoticeUnsubscribeFromObject()`	Назначение Удаляет записи вида "ресурс – маска событий", соответствующие заданному ресурсу, из приемника уведомлений, чтобы он не получал уведомления о событиях, соответствующих этим записям. Параметры

	• [in] `notice` – идентификатор приемника уведомлений. • [in] `object` – дескриптор ресурса. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки. Дополнительные сведения Уведомления, соответствующие удаленным записям вида "ресурс – маска событий", будут удалены из приемника.
`KnNoticeUnsubscribeFromEvent()`	Назначение Удаляет записи вида "ресурс – маска событий" с заданным идентификатором из приемника уведомлений, чтобы он не получал уведомления о событиях, соответствующих этим записям. Параметры • [in] `notice` – идентификатор приемника уведомлений. • [in] `evId` – идентификатор записи вида "ресурс – маска событий". Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки. Дополнительные сведения Уведомления, соответствующие удаленным записям вида "ресурс – маска событий", будут удалены из приемника.
`KnNoticeDropAndWake()`	Назначение Удаляет все записи вида "ресурс – маска событий" из заданного приемника уведомлений и возобновляет исполнение всех потоков, ожидающих появления уведомлений в заданном приемнике. Параметры • [in] `notice` – идентификатор приемника уведомлений. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.
`KnNoticeRelease()`	Назначение

	Удаляет заданный приемник уведомлений и возобновляет исполнение всех потоков, ожидающих появления уведомлений в заданном приемнике. Параметры • [in] `notice` – идентификатор приемника уведомлений. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.
`KnNoticeSetObjectEvent()`	Назначение Сигнализирует, что события, соответствующие заданной маске событий, произошли с заданным пользовательским ресурсом. Параметры • [in] `object` – дескриптор пользовательского ресурса. • [in] `evMask` – маска событий, о которых требуется сигнализировать. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.

Динамическое создание IPC-каналов (cm_api.h, ns_api.h)

API определены в заголовочных файлах из состава KasperskyOS SDK:

- `sysroot-*-kos/include/coresrv/cm/cm_api.h`;
- `sysroot-*-kos/include/coresrv/ns/ns_api.h`.

API позволяет динамически создавать IPC-каналы.

Сведения о функциях API приведены в таблицах ниже.

Использование API

Чтобы серверы могли сообщать клиентам о предоставляемых службах, нужно включить в решение сервер имен, то есть системную программу `NameServer` (исполняемый файл `sysroot-*-kos/bin/ns` из состава KasperskyOS SDK). IPC-каналы от клиентов и серверов к серверу имен можно создать статически (эти IPC-каналы должны иметь имя `kl.core.NameServer`). Если этого не сделать, то при вызове клиентами и серверами функции `NsCreate()` будут выполнены попытки динамического создания этих IPC-каналов. Сервер имен не требуется включать в решение, если у клиентов изначально есть сведения об именах серверов и предоставляемых этими серверами служб.

Имена служб и интерфейсов нужно задавать в соответствии с формальными спецификациями компонентов решения. (О квалифицированном имени службы см. "Привязка методов моделей безопасности к событиям безопасности".) Вместо квалифицированного имени службы можно использовать какое-либо условное название этой службы. Имена клиентов и серверов задаются в init-описании. Также имя процесса можно получить вызовом функции `KnTaskGetName()` из API task_api.h.

Динамическое создание IPC-канала на стороне сервера включает следующие шаги:

1. Подключиться к серверу имен вызовом функции `NsCreate()`.
2. Опубликовать предоставляемые службы на сервере имен, используя функцию `NsPublishService()`.

   Чтобы отменить публикацию службы, нужно вызвать функцию `NsUnPublishService()`.
3. Получить запрос клиента на создание IPC-канала вызовом функции `KnCmListen()`.

   Функция `KnCmListen()` позволяет получить первый запрос в очереди, но при этом не удаляет этот запрос, а помещает в конец очереди. Если в очереди всего один запрос, то вызов функции `KnCmListen()` несколько раз подряд дает один и тот же результат. Запрос удаляется из очереди при вызове функции `KnCmAccept()` или `KnCmDrop()`.
4. Принять запрос клиента на создание IPC-канала вызовом функции `KnCmAccept()`.

   Чтобы отклонить запрос клиента, нужно вызвать функцию `KnCmDrop()`.

   Слушающий дескриптор создается при вызове функции `KnCmAccept()` со значением `INVALID_HANDLE` в параметре `listener`. Если указать слушающий дескриптор, то созданный серверный IPC-дескриптор обеспечит возможность получать IPC-запросы по всем IPC-каналам, ассоциированным с этим слушающим дескриптором. (Первый IPC-канал, ассоциированный со слушающим дескриптором, создается при вызове функции `KnCmAccept()` со значением `INVALID_HANDLE` в параметре `listener`. Второй и последующие IPC-каналы, ассоциированные со слушающим дескриптором, создаются при втором и последующих вызовах функции `KnCmAccept()` с указанием дескриптора, полученного при первом вызове.) В параметре `listener` функции `KnCmAccept()` можно указать слушающий дескриптор, полученный с использованием функций `KnHandleConnect()`, `KnHandleConnectEx()` и `KnHandleCreateListener()` из API handle_api.h, а также функции `ServiceLocatorRegister()`, объявленной в заголовочном файле `sysroot-*-kos/include/coresrv/sl/sl_api.h` из состава KasperskyOS SDK.

Динамическое создание IPC-канала на стороне клиента включает следующие шаги:

1. Подключиться к серверу имен вызовом функции `NsCreate()`.
2. Найти сервер, предоставляющий требуемую службу, используя функцию `NsEnumServices()`.

   Чтобы получить полный список служб с заданным интерфейсом, нужно вызвать функцию несколько раз, инкрементируя индекс, пока не будет получена ошибка `rcResourceNotFound`.
3. Выполнить запрос на создание IPC-канала с требуемым сервером вызовом функции `KnCmConnect()`.

К серверу имен можно подключить несколько клиентов и серверов. Каждый клиент и сервер может создать несколько подключений к серверу имен. Сервер может отменить публикацию службы, выполненную другим сервером.

Удаление динамически созданных IPC-каналов

Динамически созданный IPC-канал будет удален при закрытии его клиентского и серверного IPC-дескрипторов.

Сведения о функциях API

Функции ns_api.h

Функция	Сведения о функции
`NsCreate()`	Назначение Создает подключение к серверу имен. Параметры • [in,optional] `name` – указатель на имя процесса сервера имен или `RTL_NULL`, чтобы задать имя по умолчанию (соответствует значению макроса `NS_SERVER_NAME`). • [in] `msecs` – время ожидания создания подключения к серверу имен в миллисекундах или `INFINITE_TIMEOUT`, чтобы задать неограниченное время ожидания. • [out] `ns` – указатель на идентификатор подключения к серверу имен. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.
`NsPublishService()`	Назначение Публикует службу на сервере имен. Параметры • [in] `ns` – идентификатор подключения к серверу имен. • [in] `type` – указатель на имя интерфейса службы. • [in,optional] `server` – указатель на имя сервера, предоставляющего службу, или `RTL_NULL`, чтобы использовать имя вызывающего процесса. • [in] `service` – указатель на квалифицированное имя службы. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.
`NsUnPublishService()`	Назначение Отменяет публикацию службы на сервере имен. Параметры • [in] `ns` – идентификатор подключения к серверу имен. • [in] `type` – указатель на имя интерфейса службы. • [in,optional] `server` – указатель на имя сервера, предоставляющего службу, или `RTL_NULL`, чтобы использовать имя вызывающего процесса.

	• [in] `service` – указатель на квалифицированное имя службы. Возвращаемые значения В случае успеха возвращает `rc0k`, иначе возвращает код ошибки.
`NsEnumServices()`	Назначение Перечисляет службы, опубликованные на сервере имен. Параметры • [in] `ns` – идентификатор подключения к серверу имен. • [in] `type` – указатель на имя интерфейса служб. • [in] `index` – индекс для перечисления служб. Нумерация начинается с нуля. • [out] `server` – указатель на буфер для имени сервера, предоставляющего службу. • [in] `serverSize` – размер буфера для имени сервера, предоставляющего службу, в байтах. • [out] `service` – указатель на буфер для квалифицированного имени службы. • [in] `serviceSize` – размер буфера для квалифицированного имени службы в байтах. Возвращаемые значения В случае успеха возвращает `rc0k`, иначе возвращает код ошибки.

Функции cm_api.h

Функция	Сведения о функции
`KnCmConnect()`	Назначение Выполняет запрос на создание IPC-канала с сервером для использования заданной службы. Параметры • [in] `server` – указатель на имя сервера. • [in] `service` – указатель на квалифицированное имя службы. • [in] `msecs` – время ожидания выполнения запроса в миллисекундах или `INFINITE_TIMEOUT`, чтобы задать неограниченное время ожидания. • [out] `handle` – указатель на клиентский IPC-дескриптор. • [out] `rsid` – указатель на идентификатор службы (RIID). Возвращаемые значения

	В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.
`KnCmListen()`	Назначение Позволяет получить запрос клиента на создание IPC-канала для использования службы. Параметры • [in] `filter` – фиктивный параметр, который должен иметь значение `RTL_NULL`. • [in] `msecs` – время ожидания появления запроса клиента в миллисекундах или `INFINITE_TIMEOUT`, чтобы задать неограниченное время ожидания. • [out] `client` – указатель на имя клиента. • [out] `service` – указатель на квалифицированное имя службы. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.
`KnCmDrop()`	Назначение Отклоняет запрос клиента на создание IPC-канала для использования заданной службы. Параметры • [in] `client` – указатель на имя клиента. • [in] `service` – указатель на квалифицированное имя службы. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.
`KnCmAccept()`	Назначение Принимает запрос клиента на создание IPC-канала для использования заданной службы. Параметры • [in] `client` – указатель на имя клиента. • [in] `service` – указатель на квалифицированное имя службы. • [in] `rsid` – идентификатор службы (RIID). • [in,optional] `listener` – слушающий дескриптор или `INVALID_HANDLE`, чтобы создать его. • [out] `handle` – указатель на серверный IPC-дескриптор. Возвращаемые значения

	В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.

Использование примитивов синхронизации (event.h, mutex.h, rwlock.h, semaphore.h, condvar.h)

Библиотека `libkos` предоставляет API, позволяющие использовать следующие примитивы синхронизации:

- события (`event.h`);
- мьютексы (`mutex.h`);
- блокировки чтения-записи (`rwlock.h`);
- семафоры (`semaphore.h`);
- условные переменные (`condvar.h`).

Заголовочные файлы находятся в KasperskyOS SDK по пути `sysroot-*-kos/include/kos`.

API предназначены для синхронизации потоков исполнения, принадлежащих одному и тому же процессу.

События

Событие – примитив синхронизации, который используется для уведомления одного или нескольких потоков исполнения о выполнении требуемого этим потокам условия. Уведомляемый поток исполнения ожидает, когда событие перейдет из несигнального состояния в сигнальное, а уведомляющий поток исполнения изменяет состояние этого события.

Типовой сценарий использования API для работы с событиями включает следующие шаги:

1. Инициализация события вызовом функции `KosEventInit()`.
2. Использование события потоками исполнения:
   - Ожидание перехода события из несигнального состояния в сигнальное вызовом функции `KosEventWait()` или `KosEventWaitTimeout()` (на стороне уведомляемых потоков исполнения).
   - Изменение состояния события вызовами функций `KosEventSet()` и `KosEventReset()` (на стороне уведомляющих потоков исполнения).

Сведения о функциях API для работы с событиями приведены в таблице ниже.

Функции event.h

Функция	Сведения о функции
`KosEventInit()`	Назначение Инициализирует событие. После инициализации событие находится в несигнальном состоянии. Параметры

	• [out] `event` – указатель на событие. Тип события определен в заголовочном файле `sysroot-*-kos/include/kos/sync_types.h` из состава KasperskyOS SDK. Возвращаемые значения Нет.
`KosEventSet()`	Назначение Устанавливает событие в сигнальное состояние. Параметры • [out] `event` – указатель на событие. Тип события определен в заголовочном файле `sysroot-*-kos/include/kos/sync_types.h` из состава KasperskyOS SDK. Возвращаемые значения Нет.
`KosEventReset()`	Назначение Устанавливает событие в несигнальное состояние. Параметры • [out] `event` – указатель на событие. Тип события определен в заголовочном файле `sysroot-*-kos/include/kos/sync_types.h` из состава KasperskyOS SDK. Возвращаемые значения Нет.
`KosEventWait()`	Назначение Ожидает перехода события из несигнального состояния в сигнальное сколь угодно долго. Параметры • [in,out] `event` – указатель на событие. Тип события определен в заголовочном файле `sysroot-*-kos/include/kos/sync_types.h` из состава KasperskyOS SDK. • [in] `reset` – значение, которое задает, нужно ли установить событие в несигнальное состояние после завершения ожидания (`rtl_true` – да, `rtl_false` – нет). Тип параметра определен в заголовочном файле `sysroot-*-kos/include/rtl/stdbool.h` из состава KasperskyOS SDK. Возвращаемые значения Нет.
`KosEventWaitTimeout()`	Назначение

	Ожидает перехода события из несигнального состояния в сигнальное не дольше заданного времени. Параметры • [in,out] `event` – указатель на событие. Тип события определен в заголовочном файле `sysroot-*-kos/include/kos/sync_types.h` из состава KasperskyOS SDK. • [in] `reset` – значение, которое задает, нужно ли установить событие в несигнальное состояние после завершения ожидания (`rtl_true` – да, `rtl_false` – нет). Тип параметра определен в заголовочном файле `sysroot-*-kos/include/rtl/stdbool.h` из состава KasperskyOS SDK. • [in] `mdelay` – время ожидания в миллисекундах или `INFINITE_TIMEOUT`, чтобы задать неограниченное время ожидания. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки. Если время ожидания истекло, возвращает `rcTimeout`.

Мьютексы

Мьютекс – примитив синхронизации, который обеспечивает взаимоисключающее исполнение *критических секций* (участков кода, в которых осуществляется обращение к разделяемым между потоками исполнения ресурсам). Один поток захватывает мьютекс и исполняет критическую секцию, а другие потоки, чтобы исполнить критические секции, пытаются захватить этот мьютекс, ожидая его освобождения. Мьютекс может быть освобожден только тем потоком исполнения, которым он захвачен. Можно использовать *рекурсивный мьютекс*, который может быть захвачен одним потоком исполнения несколько раз.

Типовой сценарий использования API для работы с мьютексами включает следующие шаги:

1. Инициализация мьютекса вызовом функции `KosMutexInit()` или `KosMutexInitEx()`.
2. Использование мьютекса потоками исполнения:
   a. Захват мьютекса вызовом функции `KosMutexTryLock()`, `KosMutexLock()` или `KosMutexLockTimeout()`.
   b. Освобождение мьютекса вызовом функции `KosMutexUnlock()`.

Сведения о функциях API для работы с мьютексами приведены в таблице ниже.

Функции mutex.h

Функция	Сведения о функции
`KosMutexInit()`	Назначение Инициализирует мьютекс, который не является рекурсивным. Параметры

	• [out] `mutex` – указатель на мьютекс. Тип мьютекса определен в заголовочном файле `sysroot-*-kos/include/kos/sync_types.h` из состава KasperskyOS SDK. Возвращаемые значения Нет.
`KosMutexInitEx()`	Назначение Инициализирует мьютекс. Параметры • [out] `mutex` – указатель на мьютекс. Тип мьютекса определен в заголовочном файле `sysroot-*-kos/include/kos/sync_types.h` из состава KasperskyOS SDK. • [in] `recursive` – значение, которое задает, должен ли мьютекс быть рекурсивным (`1` – да, `0` – нет). Возвращаемые значения Нет.
`KosMutexTryLock()`	Назначение Захватывает мьютекс. Если мьютекс уже захвачен, не ожидает его освобождения, а возвращает управление. Параметры • [in,out] `mutex` – указатель на мьютекс. Тип мьютекса определен в заголовочном файле `sysroot-*-kos/include/kos/sync_types.h` из состава KasperskyOS SDK. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки. Если мьютекс уже захвачен, возвращает `rcBusy`.
`KosMutexLock()`	Назначение Захватывает мьютекс. Если мьютекс уже захвачен, ожидает его освобождения сколь угодно долго. Параметры • [in,out] `mutex` – указатель на мьютекс. Тип мьютекса определен в заголовочном файле `sysroot-*-kos/include/kos/sync_types.h` из состава KasperskyOS SDK. Возвращаемые значения

	Нет.
`KosMutexUnlock()`	Назначение Освобождает мьютекс. Параметры • [in,out] `mutex` – указатель на мьютекс. Тип мьютекса определен в заголовочном файле `sysroot-*-kos/include/kos/sync_types.h` из состава KasperskyOS SDK. Возвращаемые значения Нет.
`KosMutexLockTimeout()`	Назначение Захватывает мьютекс. Если мьютекс уже захвачен, ожидает его освобождения не дольше заданного времени. Параметры • [in,out] `mutex` – указатель на мьютекс. Тип мьютекса определен в заголовочном файле `sysroot-*-kos/include/kos/sync_types.h` из состава KasperskyOS SDK. • [in] `mdelay` – время ожидания в миллисекундах или `INFINITE_TIMEOUT`, чтобы задать неограниченное время ожидания. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки. Если время ожидания истекло, возвращает `rcTimeout`.

Блокировки чтения-записи

Блокировка чтения-записи – примитив синхронизации, который используется, чтобы разрешить доступ к разделяемым между потоками исполнения ресурсам либо на запись для одного потока исполнения, либо на чтение для нескольких потоков исполнения одновременно.

Типовой сценарий использования API для работы с блокировками чтения-записи включает следующие шаги:

1. Инициализация блокировки чтения-записи вызовом функции `KosRWLockInit()`.
2. Использование блокировки чтения-записи потоками исполнения:
   a. Захват блокировки чтения-записи для записи (вызовом функции `KosRWLockWrite()` или `KosRWLockTryWrite()`) или для чтения (вызовом функции `KosRWLockRead()` или `KosRWLockTryRead()`).
   b. Освобождение блокировки-чтения вызовом функции `KosRWLockUnlock()`.

Сведения о функциях API для работы с блокировками чтения-записи приведены в таблице ниже.

Функции rwlock.h

Функция	Сведения о функции
`KosRWLockInit()`	Назначение Инициализирует блокировку чтения-записи. Параметры • [out] `rwlock` – указатель на блокировку чтения-записи. Тип блокировки чтения-записи определен в заголовочном файле `sysroot-*-kos/include/kos/sync_types.h` из состава KasperskyOS SDK. Возвращаемые значения Нет.
`KosRWLockRead()`	Назначение Захватывает блокировку чтения-записи для чтения. Если блокировка чтения-записи уже захвачена для записи, или есть потоки исполнения, ожидающие захвата этой блокировки чтения-записи для записи, то ожидает освобождения этой блокировки чтения-записи от захвата для записи сколь угодно долго. Параметры • [in,out] `rwlock` – указатель на блокировку чтения-записи. Тип блокировки чтения-записи определен в заголовочном файле `sysroot-*-kos/include/kos/sync_types.h` из состава KasperskyOS SDK. Возвращаемые значения Нет.
`KosRWLockTryRead()`	Назначение Захватывает блокировку чтения-записи для чтения. Если блокировка чтения-записи уже захвачена для записи, или есть потоки исполнения, ожидающие захвата этой блокировки чтения-записи для записи, то не ожидает освобождения этой блокировки чтения-записи от захвата для записи, а возвращает управление. Параметры • [in,out] `rwlock` – указатель на блокировку чтения-записи. Тип блокировки чтения-записи определен в заголовочном файле `sysroot-*-kos/include/kos/sync_types.h` из состава KasperskyOS SDK. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.
`KosRWLockWrite()`	Назначение

	Захватывает блокировку чтения-записи для записи. Если блокировка чтения-записи уже захвачена для записи или чтения, ожидает освобождения этой блокировки чтения-записи сколь угодно долго. Параметры • [in,out] `rwlock` – указатель на блокировку чтения-записи. Тип блокировки чтения-записи определен в заголовочном файле `sysroot-*-kos/include/kos/sync_types.h` из состава KasperskyOS SDK. Возвращаемые значения Нет.
`KosRWLockTryWrite()`	Назначение Захватывает блокировку чтения-записи для записи. Если блокировка чтения-записи уже захвачена для записи или чтения, не ожидает освобождения этой блокировки чтения-записи, а возвращает управление. Параметры • [in,out] `rwlock` – указатель на блокировку чтения-записи. Тип блокировки чтения-записи определен в заголовочном файле `sysroot-*-kos/include/kos/sync_types.h` из состава KasperskyOS SDK. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.
`KosRWLockUnlock()`	Назначение Освобождает блокировку чтения-записи. Параметры • [in,out] `rwlock` – указатель на блокировку чтения-записи. Тип блокировки чтения-записи определен в заголовочном файле `sysroot-*-kos/include/kos/sync_types.h` из состава KasperskyOS SDK. Возвращаемые значения Нет. Дополнительные сведения Если блокировка чтения-записи захвачена для чтения, то эта блокировка чтения-записи остается захваченной для чтения, пока все потоки исполнения, выполняющие чтение, не освободят ее.

Семафоры

Семафор – примитив синхронизации, основанный на счетчике, значение которого может быть атомарно изменено. Значение счетчика обычно отражает число доступных разделяемых между потоками исполнения ресурсов. Для исполнения критической секции поток ожидает, пока значение счетчика не станет больше нуля. Если значение счетчика больше нуля, то оно уменьшается на единицу, и поток исполняет критическую секцию. После исполнения критической секции поток исполнения сигнализирует семафор, в результате чего значение счетчика увеличивается.

Типовой сценарий использования API для работы с семафорами включает следующие шаги:

1. Инициализация семафора вызовом функции `KosSemaphoreInit()`.
2. Использование семафора потоками исполнения:
   a. Ожидание семафора вызовом функции `KosSemaphoreWait()`, `KosSemaphoreWaitTimeout()` или `KosSemaphoreTryWait()`.
   b. Сигнализация семафора вызовом функции `KosSemaphoreSignal()` или `KosSemaphoreSignalN()`.
3. Освобождение ресурсов семафора вызовом функции `KosSemaphoreDeinit()`.

Сведения о функциях API для работы с семафорами приведены в таблице ниже.

Функции semaphore.h

Функция	Сведения о функции
`KosSemaphoreInit()`	Назначение Инициализирует семафор. Параметры • [out] `semaphore` – указатель на семафор. Тип семафора определен в заголовочном файле `sysroot-*-kos/include/kos/sync_types.h` из состава KasperskyOS SDK. • [in] `count` – значение счетчика. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки. Если значение в параметре `count` превышает константу `KOS_SEMAPHORE_VALUE_MAX`, возвращает `rcInvalidArgument`. (Константа `KOS_SEMAPHORE_VALUE_MAX` определена в заголовочном файле `sysroot-*-kos/include/kos/sync_types.h` из состава KasperskyOS SDK.)
`KosSemaphoreDeinit()`	Назначение Освобождает ресурсы семафора. Параметры • [in] `semaphore` – указатель на семафор. Тип семафора определен в заголовочном файле `sysroot-*-kos/include/kos/sync_types.h` из состава KasperskyOS SDK.

	Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки. Если есть потоки исполнения, ожидающие семафор, возвращает `rcBusy`.
`KosSemaphoreSignal()`	Назначение Сигнализирует семафор с увеличением счетчика на единицу. Параметры • [in,out] `semaphore` – указатель на семафор. Тип семафора определен в заголовочном файле `sysroot-*-kos/include/kos/sync_types.h` из состава KasperskyOS SDK. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.
`KosSemaphoreSignalN()`	Назначение Сигнализирует семафор с увеличением счетчика на заданное число. Параметры • [in,out] `semaphore` – указатель на семафор. Тип семафора определен в заголовочном файле `sysroot-*-kos/include/kos/sync_types.h` из состава KasperskyOS SDK. • [in] `n` – натуральное число, на которое нужно увеличить счетчик. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.
`KosSemaphoreWaitTimeout()`	Назначение Ожидает семафор не дольше заданного времени. Параметры • [in,out] `semaphore` – указатель на семафор. Тип семафора определен в заголовочном файле `sysroot-*-kos/include/kos/sync_types.h` из состава KasperskyOS SDK. • [in] `mdelay` – время ожидания семафора в миллисекундах или `INFINITE_TIMEOUT`, чтобы задать неограниченное время ожидания. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки. Если время ожидания истекло, возвращает `rcTimeout`.
`KosSemaphoreWait()`	Назначение

	Ожидает семафор сколь угодно долго. Параметры • [in,out] `semaphore` – указатель на семафор. Тип семафора определен в заголовочном файле `sysroot-*-kos/include/kos/sync_types.h` из состава KasperskyOS SDK. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.
`KosSemaphoreTryWait()`	Назначение Ожидает семафор. Если счетчик семафора имеет нулевое значение, не ожидает увеличения счетчика этого семафора, а возвращает управление. Параметры • [in,out] `semaphore` – указатель на семафор. Тип семафора определен в заголовочном файле `sysroot-*-kos/include/kos/sync_types.h` из состава KasperskyOS SDK. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки. Если счетчик семафора имеет нулевое значение, возвращает `rcBusy`.

Условные переменные

Условная переменная – примитив синхронизации, который используется для уведомления одного или нескольких потоков исполнения о выполнении требуемого этим потокам условия. Условная переменная используется совместно с мьютексом. Уведомляющий и уведомляемый потоки захватывают мьютекс для исполнения критических секций. Уведомляемый поток при исполнении критической секции проверяет, выполняется ли требуемое ему условие (например, подготовлены ли данные уведомляющим потоком). Если условие выполняется, то уведомляемый поток исполняет критическую секцию и освобождает мьютекс. Если условие не выполняется, то уведомляемый поток блокируется на условной переменной, ожидая выполнения условия. При этом мьютекс автоматически освобождается. Уведомляющий поток при исполнении критической секции проверяет, выполняется ли условие, требуемое уведомляемому потоку. Если условие выполняется, то уведомляющий поток сигнализирует об этом через условную переменную и освобождает мьютекс. Уведомляемый поток, заблокированный в ожидании выполнения требуемого ему условия, возобновляет исполнение критической секции, автоматически захватывая мьютекс. После исполнения критической секции уведомляемый поток освобождает мьютекс.

Типовой сценарий использования API для работы с условными переменными включает следующие шаги:

1. Инициализация условной переменной и мьютекса.

   Чтобы инициализировать условную переменную, нужно вызвать функцию `KosCondvarInit()`.

2. Использование условной переменной и мьютекса потоками исполнения.

Использование условной переменной и мьютекса на стороне уведомляемых потоков исполнения включает следующие шаги:

1. Захват мьютекса.
2. Проверка выполнения условия.
3. Ожидание выполнения условия вызовом функции `KosCondvarWait()` или `KosCondvarWaitTimeout()`.

   После возврата функции `KosCondvarWait()` или `KosCondvarWaitTimeout()` обычно нужно снова проверить, что условие выполняется, так как другой уведомляемый поток исполнения также получил сигнал и мог сделать условие снова недействительным. (Например, другой поток мог извлечь данные, подготовленные уведомляющим потоком). Для этого нужно использовать следующую конструкцию:

   ```
   while(<условие>)
   <вызов KosCondvarWait() или KosCondvarWaitTimeout()>
   ```

4. Освобождение мьютекса.

Использование условной переменной и мьютекса на стороне уведомляющих потоков исполнения включает следующие шаги:

1. Захват мьютекса.
2. Проверка выполнения условия.
3. Сигнализация о выполнении условия вызовом функции `KosCondvarSignal()` или `KosCondvarBroadcast()`.
4. Освобождение мьютекса.

Сведения о функциях API для работы с условными переменными приведены в таблице ниже.

Функции condvar.h

Функция	Сведения о функции
`KosCondvarInit()`	Назначение Инициализирует условную переменную. Параметры • [out] `condvar` – указатель на условную переменную. Тип условной переменной определен в заголовочном файле `sysroot-*-kos/include/kos/sync_types.h` из состава KasperskyOS SDK. Возвращаемые значения Нет.
`KosCondvarWaitTimeout()`	Назначение Ожидает выполнения условия не дольше заданного времени. Параметры

	• [in] `condvar` – указатель на условную переменную. Тип условной переменной определен в заголовочном файле `sysroot-*-kos/include/kos/sync_types.h` из состава KasperskyOS SDK. • [in,out] `mutex` – указатель на мьютекс. Тип мьютекса определен в заголовочном файле `sysroot-*-kos/include/kos/sync_types.h` из состава KasperskyOS SDK. • [in] `mdelay` – время ожидания выполнения условия в миллисекундах или `INFINITE_TIMEOUT`, чтобы задать неограниченное время ожидания. Возвращаемые значения В случае успеха возвращает `rcOk`. Если время ожидания истекло, возвращает `rcTimeout`.
`KosCondvarWait()`	Назначение Ожидает выполнения условия сколь угодно долго. Параметры • [in] `condvar` – указатель на условную переменную. Тип условной переменной определен в заголовочном файле `sysroot-*-kos/include/kos/sync_types.h` из состава KasperskyOS SDK. • [in,out] `mutex` – указатель на мьютекс. Тип мьютекса определен в заголовочном файле `sysroot-*-kos/include/kos/sync_types.h` из состава KasperskyOS SDK. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.
`KosCondvarSignal()`	Назначение Сигнализирует о выполнении условия одному из потоков исполнения, ожидающих выполнения этого условия. Параметры • [in,out] `condvar` – указатель на условную переменную. Тип условной переменной определен в заголовочном файле `sysroot-*-kos/include/kos/sync_types.h` из состава KasperskyOS SDK. Возвращаемые значения Нет.
`KosCondvarBroadcast()`	Назначение Сигнализирует о выполнении условия всем потокам исполнения, ожидающим выполнения этого условия. Параметры

	• [in,out] `condvar` – указатель на условную переменную. Тип условной переменной определен в заголовочном файле `sysroot-*-kos/include/kos/sync_types.h` из состава KasperskyOS SDK. Возвращаемые значения Нет.

Управление изоляцией памяти для ввода-вывода (iommu_api.h)

API определен в заголовочном файле `sysroot-*-kos/include/coresrv/iommu/iommu_api.h` из состава KasperskyOS SDK.

API предназначен для управления изоляцией регионов физической памяти, используемых устройствами на шине PCIe для DMA. (Изоляция обеспечивается IOMMU.)

Сведения о функциях API приведены в таблице ниже.

Использование API

Без прикрепления к домену IOMMU устройство на шине PCIe не может использовать DMA. После прикрепления к домену IOMMU устройство может получить доступ ко всем буферам DMA, которые ассоциированы с этим доменом IOMMU. В один момент времени устройство может быть прикреплено только к одному домену IOMMU, но при этом к одному домену IOMMU может быть прикреплено несколько устройств. Буфер DMA может быть ассоциирован с несколькими доменами IOMMU одновременно. Каждый процесс ассоциирован с отдельным доменом IOMMU.

API позволяет прикреплять устройства на шине PCIe к домену IOMMU, ассоциированному с вызывающим процессом, и выполнять обратную операцию. Как правило, прикрепление устройства к домену IOMMU выполняется при инициализации драйвера. Открепление устройства от домена IOMMU обычно выполняется при возникновении ошибок во время инициализации драйвера или при финализации драйвера.

Ассоциация буфер DMA с доменом IOMMU создается при вызове функции `KnIoDmaBegin()`, входящей в API dma.h.

Сведения о функциях API

Функции iommu_api.h

Функция	Сведения о функции
`KnIommuAttachDevice()`	Назначение Прикрепляет устройство на шине PCIe к домену IOMMU, ассоциированному с вызывающим процессом. Параметры • [in] `bdf` – адрес устройства на шине PCIe в формате BDF. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.

	Дополнительные сведения Если IOMMU не задействован, возвращает `rcOk`.
`KnIommuDetachDevice()`	Назначение Открепляет устройство на шине PCIe от домена IOMMU, ассоциированного с вызывающим процессом. Параметры • [in] `bdf` – адрес устройства на шине PCIe в формате BDF. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки. Дополнительные сведения Если IOMMU не задействован, возвращает `rcOk`.

Использование очередей (queue.h)

API определен в заголовочном файле `sysroot-*-kos/include/kos/queue.h` из состава KasperskyOS SDK.

API позволяет организовать обмен данными между потоками исполнения, принадлежащими одному процессу, через механизм очередей без блокировок. То есть для добавления или извлечения элементов очереди не требуется блокирование других потоков исполнения, добавляющих или извлекающих элементы этой очереди.

Сведения о функциях API приведены в таблице ниже.

Использование API

Типовой сценарий использования API включает следующие шаги:

1. Создание абстракции очереди.

   Абстракция очереди состоит из структуры, содержащей метаданные об очереди, и буфера очереди, предназначенного для хранения элементов очереди. Буфер очереди логически разделен на равные участки, каждый из которых предназначен для отдельного элемента очереди. Число участков в буфере очереди соответствует максимальному числу элементов в очереди. Выравнивание адресов участков соответствует типам данных элементов очереди.

   Чтобы выполнить этот шаг, нужно вызвать функцию `KosQueueCreate()`. Эта функция может выделить память для буфера очереди или использовать уже выделенную память. Размер уже выделенной памяти должен быть достаточным, чтобы вместить максимальное число элементов в очереди. При этом нужно учитывать, что размер участка в буфере очереди округляется до ближайшего большего кратного значению выравнивания, заданному через параметр `objAlign`. Также начальный адрес уже выделенной памяти должен быть выравнен так, чтобы соответствовать типам данных элементов очереди. Если выравнивание адреса памяти, указанного в параметре `buffer`, меньше заданного через параметр `objAlign`, то функция вернет `RTL_NULL`.

2. Обмен данными между потоками исполнения через добавление и извлечение элементов очереди.

Чтобы добавить один элемент в конец очереди, нужно зарезервировать участок в буфере очереди вызовом функции `KosQueueAlloc()`, скопировать этот элемент в зарезервированный участок и вызвать функцию `KosQueuePush()`.

Чтобы добавить последовательность элементов в конец очереди, нужно зарезервировать требуемое количество участков в буфере очереди вызовами функции `KosQueueAlloc()`, скопировать элементы этой последовательности в зарезервированные участки и вызвать функцию `KosQueuePushArray()`. Порядок элементов последовательности не изменяется после добавления этой последовательности в очередь. То есть элементы добавляются очередь в том же порядке, в каком указатели на зарезервированные участки в буфере очереди помещены в массив, передаваемый через параметр `objs` функции `KosQueuePushArray()`.

Чтобы извлечь первый элемент очереди, нужно вызвать функцию `KosQueuePop()`. Эта функция возвращает указатель на зарезервированный участок в буфере очереди, который содержит первый элемент очереди. После использования извлеченного элемента (например, после проверки или сохранения значения элемента) нужно отменить резервирование занятого этим элементом участка в буфере очереди. Для этого нужно вызвать функцию `KosQueueFree()`.

Чтобы очистить очередь и отменить резервирование всех зарезервированных участков в буфере очереди, нужно вызвать функцию `KosQueueFlush()`.

3. Удаление абстракции очереди.

   Чтобы выполнить этот шаг, нужно вызвать функцию `KosQueueDestroy()`. Эта функция удаляет буфер очереди, если только память для этого буфера была выделена функцией `KosQueueCreate()`. В противном случае нужно отдельно выполнить удаление буфера очереди.

Сведения о функциях API

Функции queue.h

Функция	Сведения о функции
`KosQueueCreate()`	Назначение Создает абстракцию очереди. Параметры • [in] `objCount` – максимальное число элементов в очереди. • [in] `objSize` – размер элемента очереди в байтах. • [in] `objAlign` – выравнивание адресов участков в буфере очереди. Адреса участков в буфере очереди могут быть невыравненными (*objAlign=1*) или выравненными (*objAlign=2,4,...,2^N*) на границу *2^N*-байтовой последовательности (например, двухбайтовой, четырехбайтовой). • [in,optional] `buffer` – указатель на выделенную память для буфера очереди или `RTL_NULL`, чтобы память была выделена автоматически. Возвращаемые значения В случае успеха возвращает идентификатор абстракции очереди, иначе возвращает `RTL_NULL`.
`KosQueueDestroy()`	Назначение Удаляет абстракцию очереди.

	Параметры • [in] `queue` – идентификатор абстракции очереди. Возвращаемые значения Нет.
`KosQueueAlloc()`	Назначение Резервирует участок в буфере очереди. Параметры • [in] `queue` – идентификатор абстракции очереди. Возвращаемые значения В случае успеха возвращает указатель на зарезервированный участок в буфере очереди, иначе возвращает `RTL_NULL`.
`KosQueueFree()`	Назначение Отменяет резервирование заданного участка в буфере очереди. Параметры • [in] `queue` – идентификатор абстракции очереди. • [in] `obj` – указатель на зарезервированный участок в буфере очереди. Возвращаемые значения Нет.
`KosQueuePush()`	Назначение Добавляет элемент в конец очереди. Параметры • [in] `queue` – идентификатор абстракции очереди. • [in] `obj` – указатель на зарезервированный участок в буфере очереди. Возвращаемые значения Нет.
`KosQueuePushArray()`	Назначение Добавляет последовательность элементов в конец очереди. Параметры • [in] `queue` – идентификатор абстракции очереди.

	• [in] `objs` – массив указателей на зарезервированные участки в буфере очереди. • [in] `count` – число элементов в последовательности. Возвращаемые значения Нет.
`KosQueuePop()`	Назначение Извлекает первый элемент очереди. Параметры • [in] `queue` – идентификатор абстракции очереди. • [in] `timeout` – время ожидания появления элемента в очереди в миллисекундах или `INFINITE_TIMEOUT`, чтобы задать неограниченное время ожидания. Возвращаемые значения В случае успеха возвращает указатель на зарезервированный участок в буфере очереди, который содержит первый элемент очереди. Иначе возвращает `RTL_NULL`.
`KosQueueFlush()`	Назначение Очищает очередь и отменяет резервирование всех зарезервированных участков в буфере очереди. Параметры • [in] `queue` – идентификатор абстракции очереди. Возвращаемые значения Нет.

Использование барьеров памяти (barriers.h)

API определен в заголовочном файле `sysroot-*-kos/include/coresrv/io/barriers.h` из состава KasperskyOS SDK.

API позволяет устанавливать барьеры на чтение из памяти и/или на запись в память. *Барьер памяти* (англ. memory barrier) – это инструкция для компилятора и процессора, которая гарантирует, что операции доступа к памяти, указанные в исходном коде до установки барьера, будут выполнены до операций доступа к памяти, указанных в исходном коде после установки барьера. Использование барьеров памяти требуется, если важен порядок операций чтения из памяти и/или записи в память, поскольку действия компилятора и/или процессора, связанные с оптимизацией, могут привести к тому, что эти операции будут выполнены в порядке, отличном от указанного в исходном коде.

Сведения о функциях API приведены в таблице ниже.

Функция	Сведения о функции
`IoReadBarrier()`	Назначение Устанавливает барьер на чтение из памяти. Параметры Нет. Возвращаемые значения Нет.
`IoWriteBarrier()`	Назначение Устанавливает барьер на запись в память. Параметры Нет. Возвращаемые значения Нет.
`IoReadWriteBarrier()`	Назначение Устанавливает барьер на запись в память и чтение из памяти. Параметры Нет. Возвращаемые значения Нет.

Выполнение системных вызовов (syscalls.h)

API определен в заголовочном файле `sysroot-*-kos/include/coresrv/syscalls.h` из состава KasperskyOS SDK.

API позволяет выполнять системные вызовы `Call()`, `Recv()` и `Reply()` для отправки и получения IPC-сообщений.

Сведения о функциях API приведены в таблице ниже.

Использование API

Для передачи функциям API указателей на буферы с фиксированной частью и ареной IPC-сообщений используется заголовок IPC-сообщений, тип которого определен в заголовочном файле `sysroot-*-kos/include/ipc/if_rend.h` из состава KasperskyOS SDK. Перед вызовами функций API заголовки IPC-сообщений нужно связать с буферами, содержащими фиксированную часть и арену IPC-сообщений. Для этого нужно использовать функции `PackInMsg()` и `PackOutMsg()`, объявленные в заголовочном файле `sysroot-*-kos/include/services/rtl/nk_msg.h` из состава KasperskyOS SDK.

Функции `Call()`, `CallEx()`, `Recv()` и `RecvEx()` блокируют исполнение вызывающего потока, ожидая завершения системных вызовов. Функции `CallEx()` и `RecvEx()` позволяют задать время ожидания завершения системного вызова, по истечении которого незавершенный системный вызов прерывается, и поток, ожидающий его завершения, возобновляет исполнение. Также системный вызов прерывается, если при его выполнении возникла ошибка (например, из-за завершения серверного процесса). Завершение потока исполнения извне тоже прерывает системный вызов, завершения которого ожидает этот поток. Системный вызов, выполняемый функцией `CallEx()` или `RecvEx()`, можно прервать (например, для корректного завершения процесса) с использованием API ipc_api.h.

Если системный вызов был прерван с использованием API ipc_api.h, то функции `CallEx()` и `RecvEx()` возвращают код ошибки `rcIpcInterrupt`. Если отправка IPC-сообщения запрещена механизмами безопасности (модулем безопасности Kaspersky Security Module или механизмом безопасности на основе мандатных ссылок, реализуемым ядром KasperskyOS), то функции `Call()`, `CallEx()` и `Reply()` возвращают код ошибки `rcSecurityDisallow`.

Сведения о функциях API

Функции syscalls.h

Функция	Сведения о функции
`Call()`	Назначение Выполняет системный вызов `Call()` с неограниченным временем ожидания его завершения. Параметры • [in] `handle` – клиентский IPC-дескриптор. • [in] `msgOut` – указатель на заголовок IPC-запросов. • [in,out] `msgIn` – указатель на заголовок IPC-ответов. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.
`CallEx()`	Назначение Выполняет системный вызов `Call()` с заданным временем ожидания его завершения и возможностью прервать его выполнение. Параметры • [in] `handle` – клиентский IPC-дескриптор. • [in] `msgOut` – указатель на заголовок IPC-запросов. • [in,out] `msgIn` – указатель на заголовок IPC-ответов.

	• [in] `mdelay` – время ожидания завершения системного вызова `Call()` в миллисекундах или `INFINITE_TIMEOUT`, чтобы задать неограниченное время ожидания. • [in,optional] `syncHandle` – дескриптор объекта синхронизации IPC или `INVALID_HANDLE`, если прерывание системного вызова `Call()` не требуется. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.
`Reply()`	Назначение Выполняет системный вызов `Reply()`. Параметры • [in] `handle` – серверный IPC-дескриптор. • [in] `msgOut` – указатель на заголовок IPC-ответов. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.
`Recv()`	Назначение Выполняет системный вызов `Recv()` с неограниченным временем ожидания его завершения. Параметры • [in] `handle` – серверный IPC-дескриптор. • [in,out] `msgIn` – указатель на заголовок IPC-запросов. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.
`RecvEx()`	Назначение Выполняет системный вызов `Recv()` с заданным временем ожидания его завершения и возможностью прервать его выполнение. Параметры • [in] `handle` – серверный IPC-дескриптор. • [in,out] `msgIn` – указатель на заголовок IPC-ответов. • [in] `mdelay` – время ожидания завершения системного вызова `Recv()` в миллисекундах или `INFINITE_TIMEOUT`, чтобы задать неограниченное время ожидания. • [in,optional] `syncHandle` – дескриптор объекта синхронизации IPC или `INVALID_HANDLE`, если прерывание системного вызова `Recv()` не требуется.

	Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.

Прерывание IPC (ipc_api.h)

API определен в заголовочном файле `sysroot-*-kos/include/coresrv/ipc/ipc_api.h` из состава KasperskyOS SDK.

API позволяет прерывать системные вызовы `Call()` и `Recv()`, в ожидании завершения которых заблокирован один или несколько потоков процесса. Прерывать системные вызовы требуется, например, чтобы корректно завершить процесс, поскольку потоки, ожидающие завершения этих системных вызовов, возобновляют исполнение.

Сведения о функциях API приведены в таблице ниже.

Использование API

API позволяет прерывать системные вызовы в потоках процесса, которые заблокированы после вызова функции `CallEx()` или `RecvEx()` из API syscalls.h, если эти функции вызваны с указанием дескриптора объекта синхронизации IPC в параметре `syncHandle`. Чтобы создать объекта синхронизации IPC, нужно вызвать функцию `KnIpcCreateSyncObject()`. (Дескриптор объекта синхронизации IPC не может быть передан другому процессу, так как в маске прав этого дескриптора не установлен необходимый для этого флаг.)

Функция `KnIpcSetInterrupt()` переводит объект синхронизации IPC в состояние, при котором прерываются системные вызовы в тех потоках процесса, которые заблокированы после вызова функции `CallEx()` или `RecvEx()` с указанием дескриптора этого объекта синхронизации IPC в параметре `syncHandle`. Прерывание системного вызова возможно только на некоторых стадиях его выполнения. Системный вызов, выполняемый функцией `CallEx()`, может быть прерван только тогда, когда на сервере еще не вызвана функция `Recv()` или `RecvEx()` для того IPC-канала, клиентский IPC-дескриптор которого указан при вызове функции `CallEx()`. Системный вызов, выполняемый функцией `RecvEx()`, может быть прерван только во время ожидания IPC-запроса от клиента.

Функция `KnIpcClearInterrupt()` отменяет действие функции `KnIpcSetInterrupt()`.

Чтобы удалить объект синхронизации IPC, нужно закрыть его дескриптор вызовом функции `KnHandleClose()`, объявленной в заголовочном файле `sysroot-*-kos/include/coresrv/handle/handle_api.h` из состава KasperskyOS SDK.

Сведения о функциях API

Функции ipc_api.h

Функция	Сведения о функции
`KnIpcCreateSyncObject()`	Назначение Создает объект синхронизации IPC. Параметры

	• [out] `syncHandle` – указатель на дескриптор объекта синхронизации IPC. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.
`KnIpcSetInterrupt()`	Назначение Переводит заданный объект синхронизации IPC в состояние, при котором системные вызовы `Call()` и `Recv()` прерываются. Параметры • [in] `syncHandle` – дескриптор объекта синхронизации IPC. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.
`KnIpcClearInterrupt()`	Назначение Переводит заданный объект синхронизации IPC в состояние, при котором системные вызовы `Call()` и `Recv()` не прерываются. Параметры • [in] `syncHandle` – дескриптор объекта синхронизации IPC. Возвращаемые значения В случае успеха возвращает `rcOk`, иначе возвращает код ошибки.

Поддержка POSIX

Ограничения поддержки POSIX

В KasperskyOS ограниченно реализован POSIX с ориентацией на стандарт POSIX.1-2008. Прежде всего ограничения связаны с обеспечением безопасности.

> Отсутствует поддержка XSI и опциональной функциональности.

Ограничения затрагивают:

- взаимодействие между процессами;
- взаимодействие между потоками исполнения посредством сигналов;
- асинхронный ввод-вывод;

- использование робастных мьютексов;
- работу с терминалом;
- работу с оболочкой;
- управление дескрипторами файлов;
- использование таймеров;
- получение системных параметров.

Ограничения представлены:

- нереализованными интерфейсами;
- интерфейсами, которые реализованы с отклонениями от стандарта POSIX.1-2008;
- интерфейсами-заглушками, которые не выполняют никаких действий, кроме присвоения переменной `errno` значения `ENOSYS` и возвращения значения `-1`.

> В KasperskyOS сигналы не могут прервать системные вызовы `Call()`, `Recv()`, `Reply()`, которые обеспечивают работу библиотек, реализующих интерфейс POSIX.
> Ядро KasperskyOS не посылает сигналы.

Ограничения взаимодействия между процессами

Интерфейс	Назначение	Реализация	Заголовочный файл по стандарту POSIX.1-2008
`fork()`	Создать новый (дочерний) процесс.	Не реализован.	`unistd.h`
`pthread_atfork()`	Зарегистрировать обработчики, которые вызываются перед и после создания дочернего процесса.	Не реализован.	`pthread.h`
`wait()`	Ожидать остановки или завершения дочернего процесса.	Заглушка.	`sys/wait.h`
`waitid()`	Ожидать изменения состояния дочернего процесса.	Не реализован.	`sys/wait.h`
`waitpid()`	Ожидать	Заглушка.	`sys/wait.h`

	остановки или завершения дочернего процесса.		
`execl()`	Запустить исполняемый файл.	Заглушка.	`unistd.h`
`execle()`	Запустить исполняемый файл.	Заглушка.	`unistd.h`
`execlp()`	Запустить исполняемый файл.	Заглушка.	`unistd.h`
`execv()`	Запустить исполняемый файл.	Не реализован.	`unistd.h`
`execve()`	Запустить исполняемый файл.	Не реализован.	`unistd.h`
`execvp()`	Запустить исполняемый файл.	Заглушка.	`unistd.h`
`fexecve()`	Запустить исполняемый файл.	Заглушка.	`unistd.h`
`setpgid()`	Перевести процесс в другую группу или создать группу.	Заглушка.	`unistd.h`
`setsid()`	Создать сессию.	Заглушка.	`unistd.h`
`getpgrp()`	Получить идентификатор группы вызывающего процесса.	Заглушка.	`unistd.h`
`getpgid()`	Получить идентификатор группы.	Заглушка.	`unistd.h`
`getppid()`	Получить идентификатор родительского процесса.	Заглушка.	`unistd.h`
`getsid()`	Получить идентификатор сессии.	Заглушка.	`unistd.h`
`times()`	Получить значения времени для процесса и его потомков.	Заглушка.	`sys/times.h`
`kill()`	Послать сигнал	Можно посылать только сигнал `SIGTERM`.	`signal.h`

	процессу или группе процессов.	Параметр `pid` игнорируется.	
`pause()`	Ожидать сигнала.	Заглушка.	`unistd.h`
`sigpending()`	Проверить наличие полученных заблокированных сигналов.	Не реализован.	`signal.h`
`sigqueue()`	Послать сигнал процессу.	Не реализован.	`signal.h`
`sigtimedwait()`	Ожидать сигнала из заданного набора сигналов.	Не реализован.	`signal.h`
`sigwaitinfo()`	Ожидать сигнала из заданного набора сигналов.	Не реализован.	`signal.h`
`sem_init()`	Создать неименованный семафор.	Нельзя создать неименованный семафор для синхронизации между процессами. Если передать ненулевое значение через параметр `pshared`, то только вернет значение `-1` и присвоит переменной `errno` значение `ENOTSUP`.	`semaphore.h`
`sem_open()`	Создать/открыть именованный семафор.	Нельзя открыть именованный семафор, который был создан другим процессом. Именованные семафоры (как и неименованные) являются локальными, то есть они доступны только тому процессу, который их создал.	`semaphore.h`
`pthread_ spin_init()`	Создать спин-блокировку.	Нельзя создать спин-блокировку для синхронизации между процессами. Если передать значение `PTHREAD_PROCESS_SHARED` через параметр `pshared`, то это значение будет проигнорировано.	`pthread.h`
`mmap()`	Отобразить в память.	Нельзя выполнить отображение в память для взаимодействия между процессами. Если передать значения `MAP_SHARED` и `PROT_WRITE` через параметры `flags` и `prot` соответственно, то только вернет значение `MAP_FAILED` и присвоит переменной `errno` значение `EACCES`. Для остальных возможных значений параметра `prot` значение `MAP_SHARED` параметра `flags` игнорируется. Кроме того, через параметр `prot` нельзя передавать сочетания флагов `PROT_WRITE|PROT_EXEC` и `PROT_READ|PROT_WRITE|PROT_EXEC`. В этом случае только вернет значение `MAP_FAILED` и присвоит переменной `errno` значение `ENOMEM`.	`sys/mman.h`
`mprotect()`	Задать права доступа к памяти.	Для целей безопасности некоторые конфигурации ядра KasperskyOS запрещают предоставлять доступ к регионам	`sys/mman.h`

		виртуальной памяти на запись и исполнение одновременно. Если при использовании такой конфигурации ядра через параметр `prot` передать значение `PROT_WRITE|PROT_EXEC`, то только вернет значение `-1` и присвоит переменной `errno` значение `ENOTSUP`.	
`pipe()`	Создать неименованный канал.	Нельзя использовать неименованный канал для передачи данных между процессами. Неименованные каналы являются локальными, то есть они доступны только тому процессу, который их создал.	`unistd.h`
`mkfifo()`	Создать специальный файл FIFO (именованный канал).	Заглушка.	`sys/stat.h`
`mkfifoat()`	Создать специальный файл FIFO (именованный канал).	Не реализован.	`sys/stat.h`

Ограничения взаимодействия между потоками исполнения посредством сигналов

Интерфейс	Назначение	Реализация	Заголовочный файл по стандарту POSIX.1-2008
`pthread_kill()`	Послать сигнал потоку исполнения.	Нельзя послать сигнал потоку исполнения. Если передать номер сигнала через параметр `sig`, то только возвращается значение `ENOSYS`.	`signal.h`
`siglongjmp()`	Восстановить состояние потока управления и маску сигналов.	Не реализован.	`setjmp.h`
`sigsetjmp()`	Сохранить состояние потока управления и маску сигналов.	Не реализован.	`setjmp.h`

Ограничения асинхронного ввода-вывода

Интерфейс	Назначение	Реализация	Заголовочный файл по стандарту POSIX.1-2008
`aio_cancel()`	Отменить запросы ввода-вывода, которые ожидают обработки.	Не реализован.	`aio.h`
`aio_error()`	Получить ошибку операции асинхронного ввода-вывода.	Не реализован.	`aio.h`

`aio_fsync()`	Запросить выполнение операций ввода-вывода.	Не реализован.	`aio.h`
`aio_read()`	Запросить чтение из файла.	Не реализован.	`aio.h`
`aio_return()`	Получить статус операции асинхронного ввода-вывода.	Не реализован.	`aio.h`
`aio_suspend()`	Ожидать завершения операций асинхронного ввода-вывода.	Не реализован.	`aio.h`
`aio_write()`	Запросить запись в файл.	Не реализован.	`aio.h`
`lio_listio()`	Запросить выполнение набора операций ввода-вывода.	Не реализован.	`aio.h`

Ограничения использования робастных мьютексов

Интерфейс	Назначение	Реализация	Заголовочный файл по стандарту POSIX.1-2008
`pthread_mutex_consistent()`	Вернуть робастный мьютекс в консистентное состояние.	Не реализован.	`pthread.h`
`pthread_mutexattr_getrobust()`	Получить атрибут робастности мьютекса.	Не реализован.	`pthread.h`
`pthread_mutexattr_setrobust()`	Задать атрибут робастности мьютекса.	Не реализован.	`pthread.h`

Ограничения работы с терминалом

Интерфейс	Назначение	Реализация	Заголовочный файл по стандарту POSIX.1-2008
`ctermid()`	Получить путь к файлу управляющего терминала.	Только возвращает или передает через параметр `s` пустую строку.	`stdio.h`
`tcsetattr()`	Задать параметры терминала.	Скорость ввода, скорость вывода и другие параметры, специфичные для аппаратных терминалов, игнорируются.	`termios.h`
`tcdrain()`	Ожидать завершения вывода.	Только возвращает значение `-1`.	`termios.h`
`tcflow()`	Приостановить или возобновить прием или передачу данных.	Приостановка вывода и запуск приостановленного вывода не поддерживаются.	`termios.h`
`tcflush()`	Очистить очередь ввода или очередь вывода, или обе эти очереди.	Только возвращает значение `-1`.	`termios.h`

`tcsendbreak()`	Разорвать соединение с терминалом на заданное время.	Только возвращает значение `-1`.	`termios.h`
`ttyname()`	Получить путь к файлу терминала.	Только возвращает нулевой указатель.	`unistd.h`
`ttyname_r()`	Получить путь к файлу терминала.	Только возвращает значение ошибки.	`unistd.h`
`tcgetpgrp()`	Получить идентификатор группы процессов, использующих терминал.	Только возвращает значение `-1`.	`unistd.h`
`tcsetpgrp()`	Задать идентификатор группы процессов, использующих терминал.	Только возвращает значение `-1`.	`unistd.h`
`tcgetsid()`	Получить идентификатор группы процессов для лидера сессии, связанной с терминалом.	Только возвращает значение `-1`.	`termios.h`

Ограничения работы с оболочкой

Интерфейс	Назначение	Реализация	Заголовочный файл по стандарту POSIX.1-2008
`popen()`	Создать дочерний процесс для выполнения команды и неименованный канал с этим процессом.	Только присваивает переменной `errno` значение `ENOSYS` и возвращает значение `NULL`.	`stdio.h`
`pclose()`	Закрыть неименованный канал с дочерним процессом, созданным `popen()`, и ожидать завершения дочернего процесса.	Нельзя использовать, так как `popen()` всегда возвращает `NULL` вместо дескриптора неименованного канала, который является входными параметром для `pclose()`.	`stdio.h`
`system()`	Создать дочерний процесс для выполнения команды.	Заглушка.	`stdlib.h`
`wordexp()`	Раскрыть строку как в оболочке.	Не реализован.	`wordexp.h`
`wordfree()`	Освободить память, выделенную для результатов вызова `wordexp()`.	Не реализован.	`wordexp.h`

Ограничения управления дескрипторами файлов

Интерфейс	Назначение	Реализация	Заголовочный файл по стандарту POSIX.1-2008

`dup()`	Сделать копию дескриптора открытого файла.	Поддерживаются дескрипторы обычных файлов, стандартных потоков ввода-вывода, сокетов и каналов. Не гарантируется, что будет получен наименьший свободный дескриптор.	`fcntl.h`
`dup2()`	Сделать копию дескриптора открытого файла.	Поддерживаются дескрипторы обычных файлов, стандартных потоков ввода-вывода, сокетов и каналов. Через параметр `fildes2` нужно передавать дескриптор открытого файла.	`fcntl.h`

Ограничения использования таймеров

Интерфейс	Назначение	Реализация	Заголовочный файл по стандарту POSIX.1-2008
`clock_gettime()`	Получить значение времени.	Если передать значение `CLOCK_PROCESS_CPUTIME_ID` или `CLOCK_THREAD_CPUTIME_ID` через параметр `clock_id`, то только вернет значение `-1` и присвоит переменной `errno` значение `EINVAL`.	`time.h`
`clock()`	Получить процессорное время, затраченное на исполнение вызывающего процесса.	Возвращает время с момента запуска ядра KasperskyOS в миллисекундах.	`time.h`

Получение системных параметров

Интерфейс	Назначение	Реализация	Заголовочный файл по стандарту POSIX.1-2008
`confstr()`	Получить системный параметр.	Заглушка.	`unistd.h`

Особенности реализации POSIX

В KasperskyOS реализация некоторых интерфейсов POSIX в части, которая не определяется стандартом POSIX.1-2008, отличается от реализации этих интерфейсов в Linux и других UNIX-подобных операционных системах. Сведения об этих интерфейсах приведены в таблице ниже.

Интерфейсы POSIX с особенностями реализации

Интерфейс	Назначение	Реализация	Заголовочный файл по стандарту POSIX.1-2008

`bind()`	Назначить имя сокету.	При использовании версии VFS, которая поддерживает только сетевые операции, файлы сокетов семейства `AF_UNIX` при вызове `bind()` сохраняются в специальной файловой системе, реализуемой этой версией VFS. Файл сокета может быть создан только в корне файловой системы или в директории `/tmp`, а также может быть повторно использован после закрытия сокета.	`sys/socket.h`
`mmap()`	Отобразить в память.	На аппаратных платформах с процессорной архитектурой AArch64 (ARM64) нельзя выполнить отображение более 4 ГБ.	`sys/mman.h`
`read()`	Выполнить чтение из файла.	Если размер буфера `buf` превышает размер считанных данных, то оставшаяся часть этого буфера заполняется нулями.	`unistd.h`

Совместное использование POSIX и API libkos

В потоке исполнения, созданном с помощью Pthreads, нельзя использовать следующие API `libkos`:

- event.h, mutex.h, rwlock.h, semaphore.h, condvar.h;
- thread.h, thread_api.h;
- dma.h;
- ports.h;
- mmio.h;
- irq.h.

Следующие API `libkos` можно использовать совместно с Pthreads (и другими API POSIX):

- handle_api.h;
- notice_api.h;
- task.h, task_api.h;
- cm_api.h, ns_api.h;
- queue.h.

Интерфейсы POSIX нельзя использовать в потоках исполнения, созданных с помощью API thread.h и thread_api.h.

API syscalls.h можно использовать в любых потоках исполнения, созданных с использованием Pthreads или API thread.h и thread_api.h.

Получение статистических сведений о системе

Библиотеки `libkos` и `libc` предоставляют API для получения статистических сведений о системе. Эти сведения включают следующие пункты:

- об использовании процессорного времени системой и отдельным процессом;
- об использовании памяти системой и отдельным процессом;
- о процессах и потоках исполнения;
- о файловых системах и сетевых интерфейсах.

Получение статистических сведений о системе через API библиотеки libkos

Библиотека `libkos` предоставляет API, который позволяет получить статистические сведения об использовании процессорного времени и памяти, а также о процессах и потоках исполнения. Этот API определен в заголовочном файле `sysroot-*-kos/include/coresrv/stat/stat_api.h` из состава KasperskyOS SDK.

> API, определенный в заголовочном файле `sysroot-*-kos/include/coresrv/stat/stat_api.h` из состава KasperskyOS SDK, включает функции, которые "оборачивают" функцию `KnProfilerGetCounters()`, объявленную в заголовочном файле `sysroot-*-kos/include/coresrv/profiler/profiler_api.h` из состава KasperskyOS SDK. Эта функция запрашивает значения счетчиков производительности. Поэтому, чтобы получить статистические сведения, нужно собрать решение с версией ядра KasperskyOS, которая поддерживает счетчики производительности (подробнее см. "Библиотека image").

Получение сведений об использовании процессорного времени

Время работы процессоров (вычислительных ядер) отсчитывается с момента запуска ядра KasperskyOS.

Чтобы получить сведения об использовании процессорного времени, нужно использовать функции `KnGroupStatGetParam()`, `KnTaskStatGetParam()` и `KnCpuStatGetParam()`. При этом через параметр `param` этих функций нужно передать значения, приведенные в таблице ниже.

Сведения об использовании процессорного времени

Функция	Значение параметра `param`	Получаемое значение
`KnGroupStatGetParam()`	`GROUP_PARAM_CPU_KERNEL`	Время работы всех процессоров в режиме ядра
`KnGroupStatGetParam()`	`GROUP_PARAM_CPU_USER`	Время работы всех процессоров в пользовательском режиме
`KnGroupStatGetParam()`	`GROUP_PARAM_CPU_IDLE`	Время работы всех процессоров в режиме бездействия
`KnTaskStatGetParam()`	`TASK_PARAM_TIME_TOTAL`	Время работы всех процессоров, затраченное на исполнение заданного процесса
`KnTaskStatGetParam()`	`TASK_PARAM_TIME_USER`	Время работы всех процессоров, затраченное на исполнение заданного процесса в пользовательском режиме

`KnCpuStatGetParam()`	`CPU_STAT_PARAM_IDLE`	Время работы заданного процессора в режиме бездействия
`KnCpuStatGetParam()`	`CPU_STAT_PARAM_USER`	Время работы заданного процессора в пользовательском режиме
`KnCpuStatGetParam()`	`CPU_STAT_PARAM_KERNEL`	Время работы заданного процессора в режиме ядра

Процессорное время, полученное вызовом функции `KnGroupStatGetParam()`, `KnTaskStatGetParam()` или `KnCpuStatGetParam()`, представлено в наносекундах.

Входным параметром функции `KnCpuStatGetParam()` является индекс процессора (нумерация начинается с нуля). Чтобы получить общее число процессоров на аппаратной платформе, нужно использовать функцию `KnHalGetCpuCount()`, объявленную в заголовочном файле `sysroot-*-kos/include/coresrv/hal/hal_api.h` из состава KasperskyOS SDK.

Получение сведений об использовании памяти

Чтобы получить сведения об использовании памяти, нужно использовать функции `KnGroupStatGetParam()` и `KnTaskStatGetParam()`. При этом через параметр `param` этих функций нужно передать значения, приведенные в таблице ниже.

Сведения об использовании памяти

Функция	Значение параметра `param`	Получаемое значение
`KnGroupStatGetParam()`	`GROUP_PARAM_MEM_TOTAL`	Размер всей установленной физической памяти
`KnGroupStatGetParam()`	`GROUP_PARAM_MEM_FREE`	Размер свободной физической памяти
`KnGroupStatGetParam()`	`GROUP_PARAM_MEM_EXEC`	Размер физической памяти с атрибутом "доступ на исполнение"
`KnGroupStatGetParam()`	`GROUP_PARAM_MEM_SHARED`	Размер физической памяти, используемой в качестве разделяемой
`KnTaskStatGetParam()`	`TASK_PARAM_MEM_PHY`	Размер физической памяти, используемой заданным процессом
`KnTaskStatGetParam()`	`TASK_PARAM_MEM_VIRT`	Размер виртуальной памяти заданного процесса
`KnTaskStatGetParam()`	`TASK_PARAM_MEM_SHARED`	Размер виртуальной памяти заданного процесса, отображаемой на разделяемую физическую память
`KnTaskStatGetParam()`	`TASK_PARAM_MEM_PAGE_TABLE`	Размер таблицы страниц заданного процесса

Размер памяти, полученный вызовом функции `KnGroupStatGetParam()` или `KnTaskStatGetParam()`, представляет собой число страниц памяти. Размер страницы памяти составляет 4 КБ для всех аппаратных платформ, поддерживаемых KasperskyOS.

> Размер физической памяти, используемой процессом, характеризует только ту память, которая выделена непосредственно для этого процесса. Например, если в память процесса отображен буфер MDL, созданный другим процессом, то размер этого буфера не включается в это значение.

Получение сведений о процессах и потоках исполнения

Помимо сведений об использовании процессорного времени и памяти функции `KnGroupStatGetParam()` и `KnTaskStatGetParam()` позволяют получить сведения о процессах и потоках исполнения. Для этого через параметр `param` этих функций нужно передать значения, приведенные в таблице ниже.

Сведения о процессах и потоках исполнения

Функция	Значение параметра `param`	Получаемое значение
`KnGroupStatGetParam()`	`GROUP_PARAM_TASKS`	Число пользовательских процессов (без учета процесса ядра)
`KnGroupStatGetParam()`	`GROUP_PARAM_THREADS`	Общее число потоков исполнения (включая потоки ядра)
`KnTaskStatGetParam()`	`TASK_PARAM_PPID`	Идентификатор родительского процесса заданного процесса (PPID)
`KnTaskStatGetParam()`	`TASK_PARAM_PRIO`	Приоритет начального потока заданного процесса
`KnTaskStatGetParam()`	`TASK_PARAM_STATE`	Состояние заданного процесса (в соответствии с перечислением `TaskExecutionStates`, определенным в заголовочном файле `sysroot-*-kos/include/task/pcbpage.h` из состава KasperskyOS SDK)
`KnTaskStatGetParam()`	`TASK_PARAM_IMGSIZE`	Размер загруженного в память образа программы, исполняемой в контексте заданного процесса, в байтах
`KnTaskStatGetParam()`	`TASK_PARAM_TIME_START`	Время между запуском ядра и запуском заданного процесса в наносекундах
`KnTaskStatGetParam()`	`TASK_PARAM_HANDLES`	Число дескрипторов, принадлежащих заданному процессу
`KnTaskStatGetParam()`	`TASK_PARAM_THREADS`	Число потоков исполнения в заданном процессе

Кроме функций `KnGroupStatGetParam()` и `KnTaskStatGetParam()` для получения сведений о процессах и потоках исполнения можно использовать следующие функции:

- `KnTaskStatGetName()` позволяет получить имя заданного процесса.
- `KnTaskStatGetPath()` позволяет получить имя исполняемого файла в ROMFS, из которого запущен заданный процесс.

  Функцию можно использовать, только если процесс запущен из исполняемого файла в ROMFS. В противном случае результатом вызова функции будет пустая строка.
- `KnTaskStatGetId()` позволяет получить идентификатор заданного процесса (PID).
- `KnProfilerGetCounters()` позволяет получить значения счетчиков производительности.

  Например, чтобы получить число потоков ядра и общее число дескрипторов, нужно передать через параметр `names` значения `kl.core.Core.threads` и `handles.total`.

Получение сведений об использовании процессорного времени и памяти каждым процессом

Чтобы получить сведения об использовании процессорного времени и памяти каждым процессом, нужно выполнить следующие действия:

1. Получить список процессов вызовом функции `KnGroupStatGetTaskList()`.
2. Получить число элементов списка процессов вызовом функции `KnTaskStatGetTasksCount()`.
3. Выполнить в цикле следующие действия:
   a. Получить элемент списка процессов вызовом функции `KnTaskStatEnumTaskList()`.
   b. Получить имя процесса вызовом функции `KnTaskStatGetName()`.

      Это требуется, чтобы идентифицировать процесс, для которого будут получены сведения об использовании процессорного времени и памяти.

   c. Получить сведения об использовании процессорного времени и памяти процессом вызовами функции `KnTaskStatGetParam()`.
   d. Проверить, что процесс не завершился. Если процесс завершился, то отбросить полученные сведения об использовании процессорного времени и памяти этим процессом.

      Чтобы проверить, что процесс не завершился, нужно вызвать функцию `KnTaskStatGetParam()` с передачей через параметр `param` значения `TASK_PARAM_STATE`. Должно быть получено значение, отличное от `TaskStateTerminated`.

   e. Завершить работу с элементом списка процессов вызовом функции `KnTaskStatCloseTask()`.
4. Завершить работу со списком процессов вызовом функции `KnTaskStatCloseTaskList()`.

Расчет загрузки процессоров

Показателями загрузки процессоров (вычислительных ядер) могут быть следующие значения:

- процент загрузки всех процессоров;
- процент загрузки всех процессоров каждым процессом;
- проценты загрузки каждого процессора.

Расчет этих показателей выполняется для интервала времени, в начале и конце которого были получены сведения об использовании процессорного времени. (Например, может выполняться мониторинг загрузки процессоров с периодическим получением сведений об использовании процессорного времени.) Из значений, полученных в конце интервала, нужно вычесть значения, полученные в начале интервала. То есть для интервала нужно получить следующие приращения:

- *TK* – время работы всех процессоров в режиме ядра;
- *TKi [i=1,2,...,n]* – время работы *i*-го процессора в режиме ядра;
- *TU* – время работы всех процессоров в пользовательском режиме;
- *TUi [i=1,2,...,n]* – время работы *i*-го процессора в пользовательском режиме;

- *TIDLE* – время работы всех процессоров в режиме бездействия;
- *TIDLEi [i=1,2,...,n]* – время работы *i*-го процессора в режиме бездействия;
- *Tj [j=1,2,...,m]* – процессорное время, затраченное на исполнение *j*-го процесса.

Процент загрузки всех процессоров рассчитывается так:

(TK+TU)/(TK+TU+TIDLE).

Процент загрузки *i*-го процессора рассчитывается так:

(TKi+TUi)/(TKi+TUi+TIDLEi).

Процент загрузки всех процессоров *j*-м процессом рассчитывается так:

Tj/(TK+TU+TIDLE).

Получение статистических сведений о системе через API библиотеки libc

Библиотека `libc` предоставляет API, которые позволяют получить статистические сведения о файловых системах и сетевых интерфейсах, управляемых VFS. Функции этих API приведены в таблице ниже.

Сведения о файловых системах и сетевых интерфейсах

Функция	Заголовочный файл из состава KasperskyOS SDK	Получаемые сведения
`statvfs()`	`sysroot-*-kos/include/strict/posix/sys/statvfs.h`	Сведения о файловой системе, такие как размер блока, число блоков, число свободных блоков
`getvfsstat()`	`sysroot-*-kos/include/sys/statvfs.h`	Сведения обо всех смонтированных файловых системах, идентичные предоставляемым функцией `statvfs()`
`getifaddrs()`	`sysroot-*-kos/include/ifaddrs.h`	Сведения о сетевых интерфейсах, такие как имя, IP-адрес, маска подсети

Компонент MessageBus

Компонент `MessageBus` реализует шину сообщений, которая обеспечивает прием, распределение и доставку сообщений между программами, работающими под KasperskyOS. Шина построена по принципу "издатель-подписчик". Использование шины сообщений позволяет избежать создания большого количества IPC-каналов для связывания каждой программы-подписчика с каждой программой-издателем.

> Сообщения, передаваемые через шину `MessageBus`, не могут содержать данные. Эти сообщения могут использоваться только для уведомления подписчиков о событиях.

Компонент `MessageBus` представляет собой дополнительный уровень абстракции над KasperskyOS IPC, который позволяет упростить процесс разработки и развития ваших программ. `MessageBus` является отдельной программой, доступ к которой осуществляется через IPC, но при этом разработчикам предоставляется библиотека доступа к `MessageBus`, которая позволяет избежать использования IPC-вызовов напрямую.

API библиотеки доступа предоставляет следующие интерфейсы:

`IProviderFactory`
: Предоставляет фабричные методы для получения доступа к экземплярам остальных интерфейсов;

`IProviderControl`
: Интерфейс для регистрации и дерегистрации издателя и подписчика в шине;

`IProvider (компонент MessageBus)`
: Интерфейс для передачи сообщения в шину;

`ISubscriber`
: Интерфейс обратного вызова для передачи сообщения подписчику;

`IWaiter`
: Интерфейс ожидания обратного вызова при появлении соответствующего сообщения.

Структура сообщения

Каждое сообщение содержит два параметра:

`topic`
: Идентификатор темы сообщения;

`id`
: Дополнительный параметр, специфицирующий сообщение.

Параметры `topic` и `id` уникальны для каждого сообщения. Интерпретация `topic`+`id` определяется контрактом между издателем и подписчиком. Например, если изменяются конфигурационные данные, с которыми работают издатель и подписчик, издатель высылает сообщение об изменении данных и `id` конкретной записи с новыми данными. Подписчик, пользуясь отличными от `MessageBus` механизмами, получает новые данные по ключу `id`.

Интерфейс IProviderFactory

Интерфейс `IProviderFactory` предоставляет фабричные методы для получения интерфейсов, необходимых для работы с компонентом `MessageBus`.

Описание интерфейса `IProviderFactory` представлено в файле `messagebus/i_messagebus_control.h`.

Для получения экземпляра интерфейса `IProviderFactory` используется свободная функция `InitConnection()`, которая принимает имя IPC-канала вашей программы с программой `MessageBus`. Имя соединения задается в файле `init.yaml.in` при описании конфигурации решения. В случае успешного подключения выходной параметр содержит указатель на интерфейс `IProviderFactory`.

- Для получения интерфейса регистрации и дерегистрации (см. "Интерфейс IProviderControl") издателей и подписчиков в шине сообщений используется метод `IProviderFactory::CreateBusControl()`.

- Для получения интерфейса, содержащего методы для отправки издателем сообщений в шину (см. "Интерфейс IProvider (компонент MessageBus)"), используется метод `IProviderFactory::CreateBus()`.

- Для получения интерфейсов, содержащих методы для получения подписчиком сообщений из шины (см. "Интерфейсы ISubscriber, IWaiter и ISubscriberRunner"), используются методы `IProviderFactory::CreateCallbackWaiter` и `IProviderFactory::CreateSubscriberRunner()`.

  > Мы не рекомендуем использовать интерфейс `IWaiter`, поскольку вызов метода этого интерфейса является блокирующим.

`i_messagebus_control.h (фрагмент)`

```
class IProviderFactory
{
...
    virtual fdn::ResultCode CreateBusControl(IProviderControlPtr& controlPtr) = 0;
    virtual fdn::ResultCode CreateBus(IProviderPtr& busPtr) = 0;
    virtual fdn::ResultCode CreateCallbackWaiter(IWaiterPtr& waiterPtr) = 0;
    virtual fdn::ResultCode CreateSubscriberRunner(ISubscriberRunnerPtr& runnerPtr) =
0;
...
};
...
fdn::ResultCode InitConnection(const std::string& connectionId, IProviderFactoryPtr&
busFactoryPtr);
```

Интерфейс IProviderControl

Интерфейс `IProviderControl` предоставляет методы для регистрации и дерегистрации издателей и подписчиков в шине сообщений.

Описание интерфейса `IProviderControl` представлено в файле `messagebus/i_messagebus_control.h`.

Для получение экземпляра интерфейса используется интерфейс `IProviderFactory`.

Регистрация и дерегистрация издателя

Для регистрации издателя в шине сообщений используется метод `IProviderControl::RegisterPublisher()`. Метод принимает тему сообщения и помещает в выходной параметр уникальный идентификатор клиента шины. Если тема уже зарегистрирована в шине, то вызов будет отклонен, и идентификатор клиента не будет заполнен.

Для дерегистрации издателя в шине сообщений используется метод `IProviderControl::UnregisterPublisher()`. Метод принимает идентификатор клиента шины, полученный при регистрации. Если указан идентификатор, не зарегистрированный как идентификатор издателя, то вызов будет отклонен.

`i_messagebus_control.h (фрагмент)`

```
class IProviderControl
{
...
    virtual fdn::ResultCode RegisterPublisher(const Topic& topic, ClientId& id) = 0;
    virtual fdn::ResultCode UnregisterPublisher(ClientId id) = 0;
...
};
```

Регистрация и дерегистрация подписчика

Для регистрации подписчика в шине сообщений используется метод `IProviderControl::RegisterSubscriber()`. Метод принимает имя подписчика и список тем, на которые нужно подписаться, а в выходной параметр помещает уникальный идентификатор клиента шины.

Для дерегистрации подписчика в шине сообщений используется метод `IProviderControl::UnregisterSubscriber()`. Метод принимает идентификатор клиента шины, полученный при регистрации. Если указан идентификатор, не зарегистрированный как идентификатор подписчика, то вызов будет отклонен.

`i_messagebus_control.h (фрагмент)`

```
class IProviderControl
{
...
    virtual fdn::ResultCode RegisterSubscriber(const std::string& subscriberName,
const std::set<Topic>& topics, ClientId& id) = 0;
    virtual fdn::ResultCode UnregisterSubscriber(ClientId id) = 0;
...
};
```

Интерфейс IProvider (компонент MessageBus)

Интерфейс `IProvider` предоставляет методы для отправки издателем сообщений в шину.

Описание интерфейса `IProvider` представлено в файле `messagebus/i_messagebus.h`.

Для получение экземпляра интерфейса используется интерфейс `IProviderFactory`.

Отправка сообщения в шину

Для оправки сообщения в шину используется метод `IProvider::Push()`. Метод принимает идентификатор клиента шины, полученный при регистрации, и идентификатор сообщения. Если очередь сообщений в шине заполнена, то вызов будет отклонен.

`i_messagebus.h (фрагмент)`

```
class IProvider
{
public:
...
    virtual fdn::ResultCode Push(ClientId id, BundleId dataId) = 0;
```

```
...
};
```

Интерфейсы ISubscriber, IWaiter и ISubscriberRunner

Интерфейсы `ISubscriber`, `IWaiter` и `ISubscriberRunner` предоставляют методы для получения и обработки подписчиком сообщений из шины.

Описания интерфейсов `ISubscriber`, `IWaiter` и `ISubscriberRunner` представлено в файле `messagebus/i_subscriber.h`.

Для получение экземпляров интерфейсов `IWaiter` и `ISubscriberRunner` используется интерфейс `IProviderFactory`. Реализация callback-интерфейса `ISubscriber` предоставляется приложением-подписчиком.

Получение сообщения из шины

Чтобы перевести подписчика в режим ожидания сообщения от шины, вы можете использовать метод `IWaiter::Wait()` или `ISubscriberRunner::Run()`. Методы принимают идентификатор клиента шины и указатель на callback-интерфейс `ISubscriber`. Если идентификатор клиента не зарегистрирован, то вызов будет отклонен.

Мы не рекомендуем использовать интерфейс `IWaiter`, поскольку вызов метода `IWaiter::Wait()` является блокирующим.

При получении сообщения из шины будет вызван метод `ISubscriber::OnMessage()`. Метод принимает тему и идентификатор сообщения.

i_subscriber.h (фрагмент)

```
class ISubscriber
{
...
    virtual fdn::ResultCode OnMessage(const std::string& topic, BundleId id) = 0;
};
...
class IWaiter
{
...
    [[deprecated("Use ISubscriberRunner::Run method instead.")]]
    virtual fdn::ResultCode Wait(ClientId id, const ISubscriberPtr& subscriberPtr) =
0;
};
...
class ISubscriberRunner
{
...
    virtual fdn::ResultCode Run(ClientId id, const ISubscriberPtr& subscriberPtr) = 0;
};
```

Компонент ExecutionManager

API определен в заголовочных файлах, расположенных в директории `sysroot-*-kos/include/component/execution_manager/` из состава SDK.

Сценарий использования компонента ExecutionManager описан в статье "Запуск процесса с помощью KasperskyOS API".

Интерфейс execution_manager_proxy.h

API определен в заголовочном файле `sysroot-*-kos/include/component/execution_manager/kos_ipc/execution_manager_proxy.h`

Интерфейс содержит фабричный метод `CreateExecutionManager()` для получения указателя на экземпляр интерфейса `IExecutionManager`, необходимого для работы с компонентом ExecutionManager.

Пример использования:

client.cpp

```
#include <component/execution_manager/kos_ipc/execution_manager_proxy.h>
...
namespace execmgr = execution_manager;

int main(int argc, const char *argv[])
{
   //  ...
    execmgr::IExecutionManagerPtr ptr;
    // имя IPC-канала для соединения с процессом ExecutionManager. Должно совпадать со значением MAIN_CONN_NAME в файле CMakeLists.txt для сборки ExecutionManager.
    char mainConnection[] = "ExecMgrEntity";
    execmgr::ipc::ExecutionManagerConfig cfg{mainConnection};
    if (CreateExecutionManager(cfg, ptr) != eka::sOk)
    {
        std::cerr << "Cannot create execution manager" << std::endl;
        return EXIT_FAILURE;
    }
   // ...
}
```

Интерфейс IExecutionManager

API определен в заголовочном файле `sysroot-*-kos/include/component/execution_manager/i_execution_manager.h`

Интерфейс `IExecutionManager` позволяет получить доступ к указателям на следующие интерфейсы:

- `IApplicationController` - интерфейс для запуска\остановки процессов;
- `ISystemController` - интерфейс для управления системой.

Пример использования:

client.cpp

```
int main(int argc, const char *argv[])
{
    // ...
    execmgr::IApplicationControllerPtr ac;
    if (ptr->GetApplicationController(ac) != eka::sOk)
    {
        std::cerr << "Cannot get application controller" << std::endl;
        return EXIT_FAILURE;
    }

    execmgr::ISystemControllerPtr sc;
    if (ptr->GetSystemController(sc) != eka::sOk)
    {
        std::cerr << "Cannot get system controller" << std::endl;
        return EXIT_FAILURE;
    }

    // ...
}
```

Интерфейс IApplicationController

API определен в заголовочном файле `sysroot-*-kos/include/component/execution_manager/i_application_control.h`

Интерфейс `IApplicationController` предоставляет следующие методы, позволяющие изменять состояние процесса:

- `StartEntity(const std::filesystem::path& runPath, const StartEntityInfo& info, StartEntityResultInfo& resInfo)` - метод для запуска процесса.
- `RestartEntity(EntityId endId)` - метод для перезапуска ранее запущенного процесса.
- `ShutdownEntity(EntityId entId)` - метод для отправки процессу сигнала на завершение.
- `StopEntity(EntityId entId)` - метод для немедленной остановки исполнения процесса.

Метод `StartEntity()` принимает путь к исполняемому файлу, который нужно запустить, а также структуру с параметрами запуска процесса `StartEntityInfo`; а возвращает структуру с результатами запуска процесса `StartEntityResultInfo`. Все поля структуры `StartEntityInfo` являются опциональными для инициализации.

Остальные методы принимают структуру `EntityId`, идентифицирующую запущенный процесс.

```
struct IApplicationController
{
    // Все поля структуры StartEntityInfo являются опциональными для инициализации.
    struct StartEntityInfo
    {
        // Имя процесса. Если не указано, то будет использовано имя класса процесса.
        // Если не указано имя класса процесса, то будет использовано имя исполняемого
```

```
файла.
        std::string              entityName;
        // Класс процесса. Если не указан, то будет использовано имя процесса. Если не
указано имя процесса, то будет использовано имя исполняемого файла.
        std::string              eiid;
        std::vector<std::string> args; // Аргументы командной строки.
        std::vector<std::string> envs; // Переменные окружения.
        // Политика перезапуска процесса при его аварийном завершении. Возможные
значения:
        // EntityRestartPolicy::DoNotRestart - не перезапускать.
        // EntityRestartPolicy::AlwaysRestart - всегда перезапускать.
        EntityRestartPolicy      restartPolicy { EntityRestartPolicy::DoNotRestart };
    };

    struct StartEntityResultInfo
    {
        std::string eiid; // Класс безопасности, присвоенный процессу.
        EntityId    entId; // Структура, идентифицирующая запущенный процесс.
        Uid         sid; // Идентификатор безопасности запущенного процесса.
        std::string taskName; // Имя запущенного процесса.
    };
};
```

Пример использования:

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";

    if (ac->StartEntity(appPath, info, result) != eka::sOk)
    {
        std::cerr << "Can not start application from " << appPath << std::endl;
        return EXIT_FAILURE;
    }

    std::cout << "Application started with process sid " << result.sid << "\n";

    auto AppId = result.entId;

    if (ac->StopEntity(AppId) != eka::sOk)
    {
        std::cerr << "Cannot stop process " << appPath << std::endl;
        return EXIT_FAILURE;
    }

   // ...
}
```

Интерфейс ISystemController

API определен в заголовочном файле `sysroot-*-kos/include/component/execution_manager/i_system_control.h`

Интерфейс `ISystemController` предоставляет следующий метод для управления системой:

- `StopAllEntities()` - метод останавливает все запущенные процессы; затем завершает процесс ExecutionManager; а затем отправляет в ядро запрос на выключение устройства.

Пример использования:

client.cpp

```
int main(int argc, const char *argv[])
{
    // ...

    if (sc->StopAllEntities() != eka::sOk)
    {
        std::cerr << "Cannot stop all processes\n";
        return EXIT_FAILURE;
    }
   // ...
}
```

Сборка решения на базе KasperskyOS

Этот раздел содержит следующие сведения:

- описание процесса сборки решения на базе KasperskyOS;
- описания скриптов, библиотек и шаблонов сборки, поставляемых в KasperskyOS Community Edition;
- сведения о том, как использовать динамические библиотеки в решении на базе KasperskyOS.

Сборка образа решения

Решение на базе KasperskyOS – системное ПО (включая ядро KasperskyOS и модуль безопасности Kaspersky Security Module) и прикладное ПО, интегрированные для работы в составе программно-аппаратного комплекса.

Подробнее см. "Структура и запуск решения на базе KasperskyOS".

Системные и прикладные программы

Программы по назначению делятся на два типа:

- *Системные программы* создают инфраструктуру для прикладных программ (например, обеспечивают работу с аппаратурой, поддерживают механизм IPC, реализуют файловые системы и сетевые протоколы). Системные программы поставляются в составе KasperskyOS Community Edition. При необходимости вы можете разрабатывать собственные системные программы.
- *Прикладные программы* предназначены для взаимодействия с пользователем решения и выполнения пользовательских задач. Прикладные программы отсутствуют в составе KasperskyOS Community Edition.

Сборка программ в процессе сборки решения

При сборке решения программы делятся на два типа:

- Системные программы, поставляемые в составе KasperskyOS Community Edition в виде исполняемых файлов;
- Системные или прикладные программы, требующие компоновки в исполняемый файл.

При этом программы, требующие компоновки, делятся на следующие типы:

- Системные программы, реализующие IPC-интерфейс, для которого в составе KasperskyOS Community Edition поставляются готовые транспортные библиотеки.
- Прикладные программы, реализующие собственный IPC-интерфейс. Для их сборки необходимо генерировать транспортные методы и типы с помощью компилятора NK.
- Клиентские программы, не предоставляющие служб.

Сборка образа решения

В составе KasperskyOS Community Edition поставляются образ ядра KasperskyOS, а также исполняемые файлы некоторых системных программ и программ-драйверов, готовые к использованию в решении.

Специальная программа Einit, предназначенная для запуска всех остальных программ, а также модуль безопасности Kaspersky Security Module собираются под каждое конкретное решение и не поставляются в составе KasperskyOS Community Edition. Вместо этого в тулчейн KasperskyOS Community Edition включены утилиты для их сборки.

Общая пошаговая схема сборки описана в статье "Общая схема сборки". Сборку образа решения можно осуществлять:

- **[Рекомендовано]** при помощи скриптов системы сборки `CMake`, которые поставляются в составе KasperskyOS Community Edition.
- без использования CMake: с помощью других систем автоматизированной сборки или вручную, используя скрипты и компиляторы, поставляемые в составе KasperskyOS Community Edition.

Общая схема сборки

Для того чтобы собрать образ решения, необходимо выполнить следующие действия:

1. Подготовить EDL-, CDL- и IDL-описания прикладных программ, а также файл init-описания (по умолчанию `init.yaml`) и файлы с описанием политики безопасности решения (по умолчанию `security.psl`).

   При сборке с `CMake` EDL-описание можно генерировать используя команду `generate_edl_file()`.

2. Для всех программ, кроме системных программ, поставляемых в составе KasperskyOS Community Edition, сгенерировать файлы *.edl.h.

   - При сборке с `CMake` для этого используются команду `nk_build_edl_files()`.
   - При сборке без `CMake` для этого необходимо использовать компилятор NK.

3. Для программ, реализующих собственный IPC-интерфейс, сгенерировать код транспортных методов и типов, используемых для формирования, отправки, приема и обработки IPC-сообщений.

   - При сборке с `CMake` для этого используются команды `nk_build_idl_files()`, `nk_build_cdl_files()`.
   - При сборке без `CMake` для этого необходимо использовать компилятор NK.

4. Собрать все программы, входящие в решение, при необходимости скомпоновав их с транспортными библиотеками системных или прикладных программ. Для сборки прикладных программ, реализующих собственный IPC-интерфейс, потребуются сгенерированный на шаге 3 код, содержащий транспортные методы и типы.

   - При сборке с `CMake` для этого используются стандартные команды сборки. Необходимые настройки кросс-компиляции производятся автоматически.
   - При сборке без `CMake` для этого необходимо вручную использовать кросс-компиляторы, входящие в состав KasperskyOS Community Edition.

5. Собрать инициализирующую программу Einit.

- При сборке с `CMake` программа `Einit` собирается в процессе сборки образа решения командами `build_kos_qemu_image()` и `build_kos_hw_image()`.

- При сборке без `CMake` для генерации кода программы `Einit` необходимо использовать утилиту einit. Программу `Einit` затем необходимо собрать с помощью кросс-компилятора, поставляемого в KasperskyOS Community Edition.

6. Собрать модуль Kaspersky Security Module.

   - При сборке с `CMake` модуль безопасности собирается в процессе сборки образа решения командами `build_kos_qemu_image()` и `build_kos_hw_image()`.

   - При сборке без `CMake` для этого необходимо использовать скрипт `makekss`.

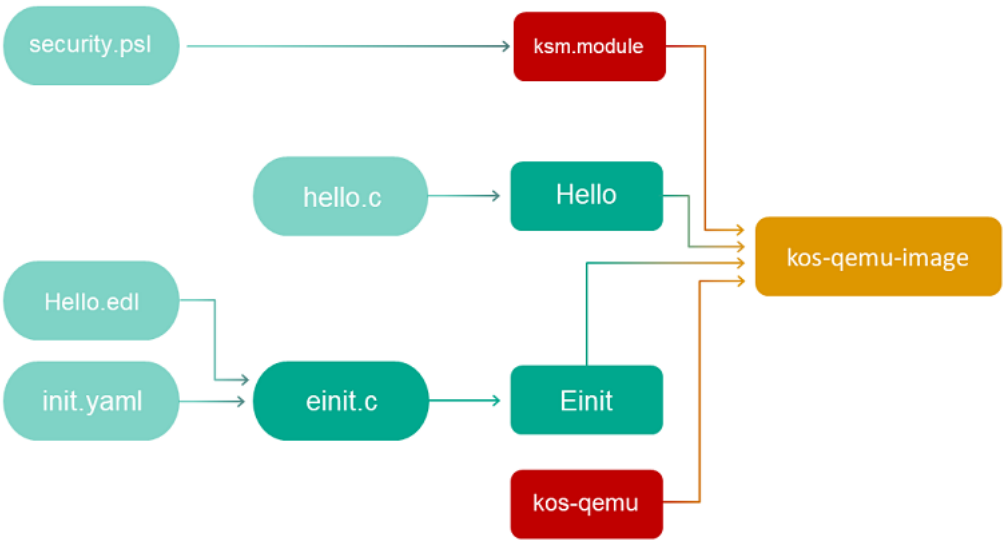
7. Создать образ решения.

   - При сборке с `CMake` для этого используются команды `build_kos_qemu_image()` и `build_kos_hw_image()`.

   - При сборке без `CMake` для этого необходимо использовать скрипт `makeimg`.

Пример 1

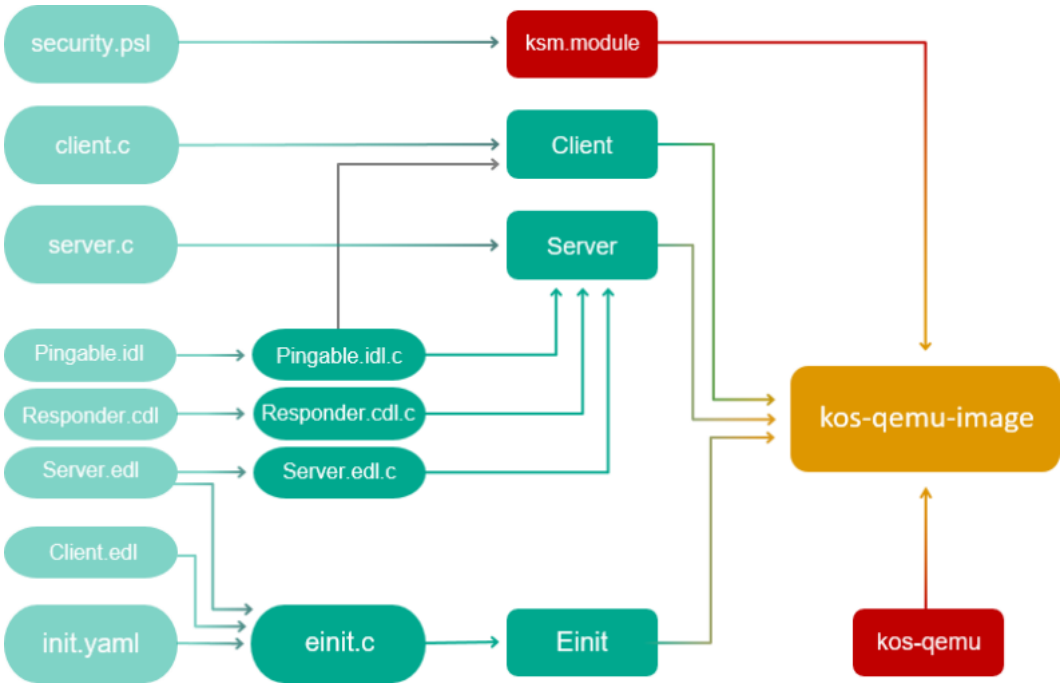
Для простейшего примера `hello`, входящего в состав KasperskyOS Community Edition, в котором содержится одна прикладная программа, не предоставляющая служб, схема сборки выглядит следующим образом:



Пример 2

Пример `echo`, входящий в состав KasperskyOS Community Edition, описывает простейший случай взаимодействия двух программ с помощью механизма IPC. Чтобы организовать такое взаимодействие, потребуется реализовать на сервере интерфейс с методом `Ping` и "поместить" службу `Ping` в новый компонент (например, `Responder`), а экземпляр этого компонента – в EDL-описание программы `Server`.

В случае наличия в решении программ, использующих механизм IPC, схема сборки выглядит следующим образом:



Использование CMake из состава KasperskyOS Community Edition

Для автоматизации процесса подготовки образа решения нужно настроить систему сборки `CMake`. За основу можно взять параметры системы сборки, используемые в примерах из состава KasperskyOS Community Edition.

В файлах `CMakeLists.txt` используется стандартный синтаксис `CMake`, а также команды и макросы из библиотек, поставляемых в KasperskyOS Community Edition.

Рекомендованная структура директорий проекта

При создании решения на базе KasperskyOS рекомендуется использовать следующую структуру директорий в проекте:

- В корне проекта создать корневой файл CMakeLists.txt, содержащий общие инструкции сборки для всего решения.
- Исходный код каждой из разрабатываемых программ следует разместить в отдельной директории, в поддиректории `src`.
- Создать файлы CMakeLists.txt для сборки каждой прикладной программы в соответствующих директориях.
- Для генерации исходного кода программы `Einit` следует создать отдельную директорию `einit`, содержащую поддиректорию `src`, в которую следует поместить шаблоны init.yaml.in и security.psl.in.

  Также в эту директорию можно поместить любые другие файлы, которые необходимо включить в образ решения.
- Создать файл CMakeLists.txt для сборки программы `Einit` в директории `einit`.
- Файлы EDL-, CDL- и IDL-описаний следует разместить в директории `resources` в корне проекта.

```
Пример структуры директорий проекта
```

```
example$ tree
.
├── CMakeLists.txt
├── hello
│   ├── CMakeLists.txt
│   ├── src
│   │   ├── hello.c
├── einit
│   ├── CMakeLists.txt
│   ├── src
│   │   ├── init.yaml.in
│   │   ├── security.psl.in
├── resources
│   ├── Hello.idl
│   ├── Hello.cdl
│   ├── Hello.edl
```

Сборка образа решения

Чтобы выполнить сборку образа решения, нужно использовать утилиту `cmake` (исполняемый файл `toolchain/bin/cmake` из состава KasperskyOS Community Edition).

Пример скрипта сборки:

build.sh

```
#!/bin/bash

# Скрипт для запуска в корне проекта.
# Сведения о параметрах запуска утилиты cmake можно
# получить shell-командой cmake --help, а также из
# официальной документации по CMake.

TARGET="aarch64-kos"
SDK_PREFIX="/opt/KasperskyOS-SDK"

# Инициализация системы сборки
cmake \
 -G "Unix Makefiles" \
 -D CMAKE_BUILD_TYPE:STRING=Release \
 -D CMAKE_TOOLCHAIN_FILE=$SDK_PREFIX/toolchain/share/toolchain-$TARGET.cmake \
 -S . \
 -B build

# Сборка
# Чтобы собрать образ решения для QEMU, нужно указать цель, заданную в
# параметре NAME CMake-команды build_kos_qemu_image() в файле CMakeLists.txt
# для сборки программы Einit.
# Чтобы собрать образ решения для аппаратной платформы, нужно указать цель,
# заданную в параметре NAME CMake-команды build_kos_hw_image() в файле
# CMakeLists.txt для сборки программы Einit.
# Чтобы собрать образ решения для QEMU и запустить QEMU с этим образом, нужно
# указать цель sim.
cmake --build build --target sim
```

Корневой файл CMakeLists.txt

Корневой файл `CMakeLists.txt` содержит общие инструкции сборки для всего решения.

Корневой файл `CMakeLists.txt` должен содержать следующие команды:

- `cmake_minimum_required (VERSION 3.25)` – указание минимальной поддерживаемой версии `CMake`.

  Для сборки решения на базе KasperskyOS требуется `CMake` версии не ниже 3.25.

  Требуемая версия `CMake` поставляется в составе KasperskyOS Community Edition и используется по умолчанию.

- `include (platform)` – подключение `CMake`-библиотеки `platform`.

- `initialize_platform()` – инициализация библиотеки `platform`.

- `project_header_default("STANDARD_GNU_11:YES" "STRICT_WARNINGS:NO")` – установка флагов компилятора и компоновщика.

- **[Опционально]** Подключение и настройка пакетов для поставляемых системных программ и драйверов, которые необходимо включить в решение:

  - Подключение пакета выполняется с помощью команды `find_package()`.

  - После подключения пакета необходимо добавить директории, связанные с этим пакетом, в список директорий поиска с помощью команды `include_directories()`.

  - Для некоторых пакетов также требуется установить значения свойств с помощью команды `set_target_properties()`.

  > `CMake`-описания системных программ и драйверов, поставляемых в составе KasperskyOS Community Edition, а также их экспортируемых переменных и свойств находятся в соответствующих файлах `/opt/KasperskyOS-Community-Edition-<version>/sysroot-aarch64-kos/lib/cmake/<имя программы>/<имя программы>-config.cmake`

- Сборка инициализирующей программы `Einit` должна быть выполнена с помощью команды `add_subdirectory(einit)`.

- Все прикладные программы, сборку которых необходимо выполнить, должны быть добавлены с помощью команды `add_subdirectory(<имя директории программы>)`.

Пример корневого файла CMakeLists.txt

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.12)
project (example)

# Инициализация библиотеки CMake для KasperskyOS SDK.
include (platform)
initialize_platform ()
```

```
project_header_default ("STANDARD_GNU_11:YES" "STRICT_WARNINGS:NO")

# Подключение пакета, импортирующего компоненты для работы с виртуальной файловой системой.
# Компоненты импортируются из папки: /opt/KasperskyOS-Community-Edition-<version>/sysroot-aarch64-kos/lib/cmake/vfs/vfs-config.cmake
find_package (vfs REQUIRED COMPONENTS ENTITY CLIENT_LIB)
include_directories (${vfs_INCLUDE})

# Подключение пакета, импортирующего компоненты для сборки программы аудита и
# подключения к ней.
find_package (klog REQUIRED)
include_directories (${klog_INCLUDE})

# Сборка инициализирующей программы Einit
add_subdirectory (einit)

# Сборка прикладной программы hello
add_subdirectory (hello)
```

Файлы CMakeLists.txt для сборки прикладных программ

Файл `CMakeLists.txt` для сборки прикладной программы должен содержать следующие команды:

- `include (platform/nk)` – подключение библиотеки `CMake` для работы с компилятором NK.
- `project_header_default ("STANDARD_GNU_11:YES" "STRICT_WARNINGS:NO")` – установка флагов компилятора и компоновщика.
- EDL-описание класса процессов для программы можно сгенерировать, используя команду `generate_edl_file()`.
- Если программа предоставляет службы, используя механизм IPC, необходимо сгенерировать транспортный код:
  a. idl.h-файлы генерируются командой `nk_build_idl_files()`
  b. cdl.h-файлы генерируются командой `nk_build_cdl_files()`
  c. edl.h-файлы генерируются командой `nk_build_edl_files()`
- `add_executable (<имя программы> "<путь к файлу исходного кода программы>")` – добавление цели для сборки программы.
- `add_dependencies (<имя программы> <имя цели сборки edl.h файла>)` – добавление зависимости сборки программы от генерации edl.h-файла.
- `target_link_libraries (<имя программы> <список библиотек>)` – определяет библиотеки, с которыми необходимо скомпоновать программу при сборке.

  Например, если программа использует файловый или сетевой ввод/вывод, то она должна быть скомпонована с транспортной библиотекой `${vfs_CLIENT_LIB}`.

> `CMake`-описания системных программ и драйверов, поставляемых в составе KasperskyOS Community Edition, а также их экспортированных переменных и свойств находятся в соответствующих файлах `/opt/KasperskyOS-Community-Edition-<version>/sysroot-aarch64-kos/lib/cmake/<имя программы>/<имя программы>-config.cmake`

- Для автоматического добавления описаний IPC-каналов в файл `init.yaml` при сборке решения необходимо определить свойство `EXTRA_CONNECTIONS` и присвоить ему значение с описаниями нужных IPC-каналов.

  Обратите внимание на отступы в начале строк в свойстве `EXTRA_CONNECTIONS`. Эти отступы необходимы для корректной подстановки значений в файл `init.yaml` и должны соответствовать требованиям к его синтаксису.

  Пример создания IPC-канала между процессами `Client` и `Server`:

  ```
  set_target_properties (Client PROPERTIES
  EXTRA_CONNECTIONS
  "  - target: Server
      id: server_connection")
  ```

  В результате, при сборке решения, описание этого IPC-канала будет автоматически добавлено в файл `init.yaml` на этапе обработки макросов шаблона init.yaml.in.

- Для автоматического добавления списка аргументов функции `main()` и словаря переменных окружения в файл `init.yaml` при сборке решения, необходимо определить свойства `EXTRA_ARGS` и `EXTRA_ENV` и присвоить им соответствующие значения.

  Обратите внимание на отступы в начале строк в свойствах `EXTRA_ARGS` и `EXTRA_ENV`. Эти отступы необходимы для корректной подстановки значений в файл `init.yaml` и должны соответствовать требованиям к его синтаксису.

  Пример передачи программе `Client` аргумента `"-v"` функции `main()` и переменной окружения `VAR1` со значением `VALUE1`:

  ```
  set_target_properties (Client PROPERTIES
  EXTRA_ARGS
  "    - \"-v\""
  EXTRA_ENV
  "    VAR1: VALUE1")
  ```

  В результате, при сборке решения, описание аргумента функции `main()` и значение переменной окружения будут автоматически добавлены в файл `init.yaml` на этапе обработки макросов шаблона init.yaml.in.

Пример файла CMakeLists.txt для сборки простой прикладной программы

CMakeLists.txt

```
project (hello)

# Инструментарий для работы с компилятором NK.
include (platform/nk)

# Установка флагов компиляции.
project_header_default ("STANDARD_GNU_11:YES" "STRICT_WARNINGS:NO")
```

```
# Задаем имя проекта, в который входит программа.
set (LOCAL_MODULE_NAME "example")

# Задаем имя программы.
set (ENTITY_NAME "Hello")
# Обратите внимание на содержание шаблонов init.yaml.in и security.psl.in
# В них имена программ задаются как ${LOCAL_MODULE_NAME}.${ENTITY_NAME}

# Задаем цели, которые будут использованы для создания генерируемых файлов программы.
set (ENTITY_IDL_TARGET ${ENTITY_NAME}_idl)
set (ENTITY_CDL_TARGET ${ENTITY_NAME}_cdl)
set (ENTITY_EDL_TARGET ${ENTITY_NAME}_edl)

# Задаем имя цели, которая будет использована для построения программы.
set (APP_TARGET ${ENTITY_NAME}_app)

# Добавляем цель сборки idl.h-файла.
nk_build_idl_files (${ENTITY_IDL_TARGET}
                    NK_MODULE ${LOCAL_MODULE_NAME}
                    IDL "resources/Hello.idl"
                    )

# Добавляем цель сборки cdl.h-файла.
nk_build_cdl_files (${ENTITY_CDL_TARGET}
                    IDL_TARGET ${ENTITY_IDL_TARGET}
                    NK_MODULE ${LOCAL_MODULE_NAME}
                    CDL "resources/Hello.cdl")

# Добавляем цель сборки EDL-файла. Переменная EDL_FILE экспортируется
# и содержит путь до сгенерированного EDL-файла.
generate_edl_file (  ${ENTITY_NAME}
                    PREFIX ${LOCAL_MODULE_NAME}
                   )

# Добавляем цель сборки edl.h-файла.
nk_build_edl_files (${ENTITY_EDL_TARGET}
                    NK_MODULE ${LOCAL_MODULE_NAME}
                    EDL ${EDL_FILE}
                    )

# Определяем цель для сборки программы.
add_executable (${APP_TARGET} "src/hello.c")
# Имя программы в init.yaml и security.psl и имя исполняемого файла должны совпадать
set_target_properties (${APP_TARGET} PROPERTIES OUTPUT_NAME ${ENTITY_NAME})
# Библиотеки, с которыми программа компонуется при сборке
target_link_libraries ( ${APP_TARGET}
                        PUBLIC  ${vfs_CLIENT_LIB}   # Программа использует файловый
ввод/вывод
                                                    # и должна быть подключена как
клиент к VFS
                        )
```

Файл CMakeLists.txt для сборки программы Einit

Файл `CMakeLists.txt` для сборки инициализирующей программы `Einit` должен содержать следующие команды:

- `include (platform/image)` – подключение библиотеки `CMake`, содержащей скрипты сборки образа решения.
- `project_header_default ("STANDARD_GNU_11:YES" "STRICT_WARNINGS:NO")` – установка флагов компилятора и компоновщика.
- Настройка пакетов системных программ и драйверов, которые необходимо включить в решение.
  - Подключение пакета выполняется с помощью команды `find_package ()`.
  - Для некоторых пакетов также требуется установить значения свойств с помощью команды `set_target_properties ()`.

> `CMake`-описания системных программ и драйверов, поставляемых в составе KasperskyOS Community Edition, а также их экспортированных переменных и свойств находятся в соответствующих файлах `/opt/KasperskyOS-Community-Edition-<version>/sysroot-aarch64-kos/lib/cmake/<имя программы>/<имя программы>-config.cmake`

- Для автоматического добавления описаний IPC-каналов между процессами системных программ в файл `init.yaml` при сборке решения необходимо добавить эти каналы в свойство `EXTRA_CONNECTIONS` для соответствующих программ.

  Обратите внимание на отступы в начале строк в свойстве `EXTRA_CONNECTIONS`. Эти отступы необходимы для корректной подстановки значений в файл `init.yaml` и должны соответствовать требованиям к его синтаксису.

  Например, программа `VFS` по умолчанию не имеет канала для соединения с программой `Env`. Чтобы описание такого канала автоматически добавилось в файл init.yaml при сборке решения, необходимо добавить следующий вызов в файл `CMakeLists.txt` для сборки программы `Einit`:

  ```
  set_target_properties (${vfs_ENTITY} PROPERTIES
  EXTRA_CONNECTIONS
  "   - target: env.Env
       id: {var: ENV_SERVICE_NAME, include: env/env.h}"
  ```

  В результате, при сборке решения, описание этого IPC-канала будет автоматически добавлено в файл `init.yaml` на этапе обработки макросов шаблона init.yaml.in.

- Для автоматического добавления списка аргументов функции `main()` и словаря переменных окружения в файл `init.yaml` при сборке решения, необходимо определить свойства `EXTRA_ARGS` и `EXTRA_ENV` и присвоить им соответствующие значения.

  Обратите внимание на отступы в начале строк в свойствах `EXTRA_ARGS` и `EXTRA_ENV`. Эти отступы необходимы для корректной подстановки значений в файл `init.yaml` и должны соответствовать требованиям к его синтаксису.

Пример передачи программе `VfsEntity` аргумента `"-f fstab"` функции `main()` и переменной окружения `ROOTFS` со значением `ramdisk0,0 / ext2 0`:

```
set_target_properties (${vfs_ENTITY} PROPERTIES
EXTRA_ARGS
"    - \"-f\"
   - \"fstab\""
EXTRA_ENV
"    ROOTFS: ramdisk0,0 / ext2 0")
```

В результате, при сборке решения, описание аргумента функции `main()` и значение переменной окружения будут автоматически добавлены в файл `init.yaml` на этапе обработки макросов шаблона init.yaml.in.

- `set(ENTITIES <полный список программ, входящих в решение>)` – определение переменной `ENTITIES` со списком исполняемых файлов всех программ, входящих в решение.

- Одна или обе команды для сборки образа решения:

  - build_kos_hw_image() – создает цель сборки образа решения для аппаратной платформы.

  - build_kos_qemu_image() – создает цель сборки образа решения для QEMU.

Пример файла CMakeLists.txt для сборки программы Einit

CMakeLitsts.txt

```
project (einit)

# Подключение библиотеки, содержащей скрипты сборки образа решения.
include (platform/image)

# Установка флагов компиляции.
project_header_default ("STANDARD_GNU_11:YES" "STRICT_WARNINGS:NO")

# Настройка программы VFS.
# По умолчанию программе VFS не сопоставляется программа, реализующая блочное
устройство.
# Если необходимо использовать блочное устройство, например ata из компонента ata,
# необходимо задать это устройство в переменной ${blkdev_ENTITY}_REPLACEMENT
# Больше информации об экспортированных переменных и свойств программы VFS
# см. в /opt/KasperskyOS-Community-Edition-<version>/sysroot-aarch64-
kos/lib/cmake/vfs/vfs-config.cmake
# find_package(ata)
# set_target_properties (${vfs_ENTITY} PROPERTIES ${blkdev_ENTITY}_REPLACEMENT
${ata_ENTITY})
# В простейшем случае не нужно взаимодействовать с диском,
# поэтому мы устанавливаем значение переменной ${blkdev_ENTITY}_REPLACEMENT равным
пустой строке
set_target_properties (${vfs_ENTITY} PROPERTIES ${blkdev_ENTITY}_REPLACEMENT "")

# Определение переменной ENTITIES со списком исполняемых файлов программ
# Важно включить все программы, входящие в проект, кроме программы Einit.
# Обратите внимание на то, что имя исполняемого файла программы должно
# совпадать с названием цели, указанной в add_executable() в CMakeLists.txt для сборки
этой программы.
set(ENTITIES
    ${vfs_ENTITY}
    Hello_app
    )

# Создание цели сборки с именем kos-image, которая является образом решения для
аппаратной платформы.
build_kos_hw_image (kos-image
                    EINIT_ENTITY EinitHw
                    CONNECTIONS_CFG "src/init.yaml.in" # шаблон файла init.yaml
                    SECURITY_PSL "src/security.psl.in" # шаблон файла security.psl
                    IMAGE_FILES ${ENTITIES}
                    )
```

```
# Создание цели сборки с именем kos-qemu-image, которая является образом решения для
QEMU.
build_kos_qemu_image (kos-qemu-image
                      EINIT_ENTITY EinitQemu
                      CONNECTIONS_CFG "src/init.yaml.in"
                      SECURITY_PSL "src/security.psl.in"
                      IMAGE_FILES ${ENTITIES}
                      )
```

Шаблон init.yaml.in

Шаблон `init.yaml.in` используется для автоматической генерации *части* файла `init.yaml` перед сборкой программы `Einit` средствами `CMake`.

Использование шаблона `init.yaml.in` позволяет не добавлять описания системных программ и IPC-каналов для соединения с ними в файл `init.yaml` вручную.

Шаблон `init.yaml.in` должен содержать следующие данные:

- Корневой ключ `entities`.
- Список всех прикладных программ, входящих в решение.
- Для прикладных программ, использующих механизм IPC, необходимо указать список IPC-каналов, соединяющих эту программу с другими программами.

  IPC-каналы, соединяющие эту программу с другими *прикладными* программами указываются вручную или в файле CMakeLists.txt этой программы с помощью свойства `EXTRA_CONNECTIONS`.

  Для указания списка IPC-каналов, соединяющих эту программу с системными программами, входящими в состав KasperskyOS Community Edition, используются следующие макросы:

  - `@INIT_<имя программы>_ENTITY_CONNECTIONS@` – при сборке заменяется на список IPC-каналов со всеми системными программами, с которыми скомпонована прикладная программа. Поля `target` и `id` заполняются в соответствии с файлами `connect.yaml` из состава KasperskyOS Community Edition, расположенными в `/opt/KasperskyOS-Community-Edition-<version>/sysroot-aarch64-kos/include/<имя системной программы>`).

    Этот макрос нужно использовать, если прикладная программа не имеет соединений с другими прикладными программами и соединяется *только с системными программами*. Этот макрос *добавляет* корневой ключ `connections`.

  - `@INIT_<имя программы>_ENTITY_CONNECTIONS+@` – при сборке добавляет список IPC-каналов со всеми системными программами, с которыми скомпонована прикладная программа, к списку IPC-каналов, заданному вручную. Этот макрос *не добавляет* корневой ключ `connections`.

    Этот макрос нужно использовать, если прикладная программа имеет соединения с другими прикладными программами, которые были указаны в шаблоне `init.yaml.in` вручную.

- Макросы `@INIT_<имя программы>_ENTITY_CONNECTIONS@` и `@INIT_<имя программы>_ENTITY_CONNECTIONS+@` также добавляют список соединений для каждой программы, заданный в свойстве `EXTRA_CONNECTIONS` при сборке этой программы.
- Если необходимо передать программе аргументы функции `main()`, заданные в свойстве `EXTRA_ARGS` при сборке этой программы, то необходимо использовать следующие макросы:

    - `@INIT_<имя программы>_ENTITY_ARGS@` – при сборке заменяется на список аргументов функции `main()`, заданный в свойстве `EXTRA_ARGS`. Этот макрос *добавляет* корневой ключ `args`.

    - `@INIT_<имя программы>_ENTITY_ARGS+@` – при сборке добавляет список аргументов функции `main()`, заданный в свойстве `EXTRA_ARGS`, к списку аргументов заданному вручную. Этот макрос *не добавляет* корневой ключ `args`.

- Если необходимо передать программе значения переменных окружения, заданные в свойстве `EXTRA_ENV` при сборке этой программы, то необходимо использовать следующие макросы:

    - `@INIT_<имя программы>_ENTITY_ENV@` – при сборке заменяется на словарь переменных окружения и их значений, заданный в свойстве `EXTRA_ENV`. Этот макрос *добавляет* корневой ключ `env`.

    - `@INIT_<имя программы>_ENTITY_ENV+@` – при сборке добавляет словарь переменных окружения и их значений, заданный в свойстве `EXTRA_ENV`, к переменным заданным вручную. Этот макрос *не добавляет* корневой ключ `env`.

- Макрос `@INIT_EXTERNAL_ENTITIES@`, который при сборке заменяется на список системных программ, с которыми скомпонована прикладная программа, и их IPC-каналов, аргументов функции `main()` и значений переменных окружения.

Пример шаблона init.yaml.in

init.yaml.in

```
entities:

- name: ping.Client
  connections:
  # Программа "Client" может обращаться к "Server".
  - target: ping.Server
    id: server_connection
@INIT_Client_ENTITY_CONNECTIONS+@
@INIT_Client_ENTITY_ARGS@
@INIT_Client_ENTITY_ENV@

- name: ping.Server
@INIT_Server_ENTITY_CONNECTIONS@

@INIT_EXTERNAL_ENTITIES@
```

При сборке программы `Einit` из этого шаблона будет сгенерирован следующий файл init.yaml:

init.yaml

```
entities:

- name: ping.Client
  connections:
  # Программа "Client" может обращаться к "Server"
  - target: ping.Server
    id: server_connection
  - target: kl.VfsEntity
    id: {var: _VFS_CONNECTION_ID, include: vfs/defs.h}
  args:
  - "-v"
  env:
```

```
    VAR1: VALUE1

- name: ping.Server
  connections:
  - target: kl.VfsEntity
    id: {var: _VFS_CONNECTION_ID, include: vfs/defs.h}

- name: kl.VfsEntity
  path: VFS
  args:
  - "-f"
  - "fstab"
  env:
    ROOTFS: ramdisk0,0 / ext2
```

Шаблон security.psl.in

Шаблон `security.psl.in` используется для автоматической генерации *части* файла `security.psl` перед сборкой программы `Einit` средствами `CMake`.

Файл `security.psl` содержит часть описания политики безопасности решения.

Использование шаблона `security.psl.in` позволяет не добавлять EDL-описания системных программ в файл `security.psl` вручную.

Шаблон `security.psl.in` должен содержать описание политики безопасности решения, созданное вручную, включая следующие декларации:

- установка глобальных параметров политики безопасности решения;
- включение PSL-файлов в описание политики безопасности решения;
- включение EDL-файлов прикладных программ в описание политики безопасности решения;
- создание объектов моделей безопасности;
- привязка методов моделей безопасности к событиям безопасности;
- создание профилей аудита безопасности.

Для автоматического включения системных программ, необходимо использовать макрос `@INIT_EXTERNAL_ENTITIES@`.

Пример шаблона security.psl.in

security.psl.in

```
execute: kl.core.Execute

use nk.base._

use EDL Einit
use EDL kl.core.Core
```

```
use EDL Client
use EDL Server
@INIT_EXTERNAL_ENTITIES@

/* Запуск программ разрешен */
execute {
    grant ()
}
/* Отправка и получение запросов, ответов и ошибок разрешены. */
request {
    grant ()
}
response {
    grant ()
}

error {
    grant ()
}
/* Обращения по интерфейсу безопасности игнорируются. */
security {
    grant ()
}
```

Библиотеки CMake в составе KasperskyOS Community Edition

Этот раздел содержит описание библиотек, поставляемых в KasperskyOS Community Edition и предназначенных для автоматизации сборки решения на базе KasperskyOS.

Библиотека platform

Библиотека `platform` содержит следующие команды:

- `initialize_platform()` – команда для инициализации библиотеки `platform`.

  Команда `initialize_platform()` может вызываться с параметром `FORCE_STATIC`, который включает принудительную статическую компоновку исполняемых файлов:

  - По умолчанию, если тулчейн в составе KasperskyOS SDK поддерживает динамическую компоновку, то команда `initialize_platform()` делает так, что для сборки всех исполняемых файлов, заданных через `CMake`-команды `add_executable()`, флаг `-rdynamic` используется автоматически.
  - При вызове `initialize_platform (FORCE_STATIC)` в корневом файле `CMakeLists.txt` тулчейн, поддерживающий динамическую компоновку, выполняет статическую компоновку исполняемых файлов.

  Команда `initialize_platform()` может вызываться с параметром `NO_NEW_VERSION_CHECK`, который отключает проверку наличия обновлений SDK и передачу версии SDK на сервер "Лаборатории Касперского".

  Чтобы отключить проверку наличия обновлений SDK и передачу данных версии SDK на сервер Kaspersky при сборке решения используйте следующий вызов: `initialize_platform(NO_NEW_VERSION_CHECK)`. Подробнее о политике предоставления данных см. "Предоставление данных".

- `project_static_executable_header_default()` – команда для включения принудительной статической компоновки исполняемых файлов, заданных через последующие `CMake`-команды `add_executable()` в одном файле `CMakeLists.txt`. Тулчейн, поддерживающий динамическую компоновку, выполняет статическую компоновку этих исполняемых файлов.

- `platform_target_force_static()` – команда для включения принудительной статической компоновки исполняемого файла, заданного через `CMake`-команду `add_executable()`. Тулчейн, поддерживающий динамическую компоновку, выполняет статическую компоновку этого исполняемого файла. Например, если вызываются `CMake`-команды `add_executable(client "src/client.c")` и `platform_target_force_static(client)`, то для программы `client` выполняется статическая компоновка.

- `project_header_default()` – команда для указания флагов компиляции.

  Параметры команды задаются в виде пар, состоящих из флага компиляции и его значения: `"FLAG_1:VALUE_1"` `"FLAG_2:VALUE_2"` … `"FLAG_N:VALUE_N"`. `CMake`-библиотека `platform` преобразует эти пары в параметры компилятора. Часто используемые флаги компиляции для компиляторов C и C++ из набора GCC, а также значения этих флагов приведены в таблице ниже.

Флаг компиляции	Значение `YES`	Значение `NO`	Значение по умолчанию
STANDARD_ANSI	Используются стандарты ISO C90 и 1998 ISO C++. Для компиляторов C и C++ значение преобразуется в параметр: `-ansi`.	Стандарты ISO C90 и 1998 ISO C++ не используются.	`STANDARD_ANSI:NO`
STANDARD_C99	Используется стандарт ISO C99. Для компилятора C значение преобразуется в параметр: `-std=c99`.	Стандарт ISO C99 не используется.	`STANDARD_C99:NO`
STANDARD_GNU_C99	Используется стандарт ISO C99 с расширениями GNU. Для компилятора C значение преобразуется в параметр: `-std=gnu99`.	Стандарт ISO C99 с расширениями GNU не используется.	`STANDARD_GNU_C99:NO`
STANDARD_11	Используются стандарты ISO C11 и 2011 ISO C++. Для C-компилятора значение преобразуется в параметр `-std=c11` или `-std=c1x` в зависимости от версии компилятора. Для компилятора C++ значение преобразуется в параметр `-std=c++11` или `-std=c++0x` в зависимости от версии компилятора.	Стандарты ISO C11 и 2011 ISO C++ не используются.	`STANDARD_11:NO`
STANDARD_GNU_11	Используются стандарты ISO C11 и 2011 ISO C++ с расширениями GNU.	Стандарты ISO C11 и 2011 ISO C++ с расширениями GNU не используются.	`STANDARD_GNU_11:NO`

	Для C-компилятора значение преобразуется в параметр `-std=gnu1x` или `-std=gnu11` в зависимости от версии компилятора. Для компилятора C++ значение преобразуется в параметр `-std=gnu++0x` или `-std=gnu++11` в зависимости от версии компилятора.		
STANDARD_14	Используется стандарт 2014 ISO C++. Для компилятора C++ значение преобразуется в параметр `-std=c++14`.	Стандарт 2014 ISO C++ не используется.	`STANDARD_14:NO`
STANDARD_GNU_14	Используется стандарт 2014 ISO C++ с расширениями GNU. Для компилятора C++ значение преобразуется в параметр `-std=gnu++14`.	Стандарт 2014 ISO C++ с расширениями GNU не используется.	`STANDARD_GNU_14:NO`
STANDARD_17	Используются стандарты ISO C17 и 2017 ISO C++. Для C-компилятора значение преобразуется в параметр `-std=c17`. Для компилятора C++ значение преобразуется в параметр `-std=c++17`.	Стандарты ISO C17 и 2017 ISO C++ не используются.	`STANDARD_17:NO`
STANDARD_GNU_17	Используются стандарты ISO C17 и 2017 ISO C++ с расширениями GNU. Для C-компилятора значение преобразуется в параметр `-std=gnu17`. Для компилятора C++ значение преобразуется в параметр `-std=gnu++17`.	Стандарты ISO C17 и 2017 ISO C++ с расширениями GNU не используются.	`STANDARD_GNU_17:NO`
STRICT_WARNINGS	Включены предупреждения для обнаружения потенциальных проблем и ошибок в коде на языках C и C++. Для компиляторов C и C++ значение преобразуется в следующие параметры: `-Wcast-qual`, `-Wcast-align`, `-Wundef`.	Предупреждения отключены.	`STRICT_WARNINGS:YES`

	Для компилятора C дополнительно используется параметр `-Wmissing-prototypes`.		

Если через параметры команды не заданы флаги компиляции вида `STANDART_*`, то по умолчанию используется параметр `STANDARD_GNU_17:YES`.

При использовании команд `initialize_platform(FORCE_STATIC)`, `project_static_executable_header_default()` и `platform_target_force_static()` могут возникать ошибки компоновки, если статический вариант требуемых библиотек отсутствует (например, не собран или не поставлен в составе KasperskyOS SDK). Но даже при наличии статического варианта требуемых библиотек эти ошибки могут возникать из-за того, что при использовании команд `initialize_platform(FORCE_STATIC)`, `project_static_executable_header_default()` и `platform_target_force_static()` система сборки по умолчанию может выполнять поиск динамического варианта требуемых библиотек, а не статического, как ожидается. Чтобы избежать ошибок, нужно, во-первых, обеспечить наличие статического варианта требуемых библиотек, во-вторых, настроить систему сборки на поиск статических библиотек (для некоторых библиотек этой возможности может не быть) либо явно задавать компоновку со статическими библиотеками.

Примеры настройки системы сборки на поиск статических библиотек:

```
set (fmt_USE_STATIC ON)
find_package (fmt REQUIRED)

set (fdn_USE_STATIC ON)
find_package (fdn REQUIRED)

set (sqlite_wrapper_USE_STATIC ON)
find_package (sqlite_wrapper REQUIRED)
```

Пример, в котором явно задана компоновка со статической библиотекой:

```
target_link_libraries(${PROJECT_NAME} PUBLIC logger::logger-static)
```

Подробнее об использовании динамических библиотек см. "Использование динамических библиотек".

Эти команды используются в файлах `CmakeLists.txt` для программы Einit и прикладных программ.

Библиотека nk

Этот раздел содержит описание команд и макросов библиотеки `CMake`-библиотеки для работы с компилятором NK.

generate_edl_file()

Команда объявлена в файле `/opt/KasperskyOS-Community-Edition-<version>toolchain/share/cmake/Modules/platform/nk2.cmake`.

```
generate_edl_file(NAME ...)
```

Команда генерирует EDL-файл с описанием класса процессов.

Параметры:

- `NAME` – имя создаваемого EDL-файла. Обязательный параметр.
- `PREFIX` – в этом параметре необходимо указать имя класса процессов, исключив из него имя EDL-файла. Например, если имя класса процессов, для которого создается EDL-файл, определено как `kl.core.NameServer`, то в параметре `PREFIX` необходимо передать значение `kl.core`.
- `EDL_COMPONENTS` – имя компонента и его экземпляра, которые будут включены в EDL-файл. Например: `EDL_COMPONENTS "env: kl.Env"`. Для включения нескольких компонентов нужно использовать несколько параметров `EDL_COMPONENTS`.
- `SECURITY` – квалифицированное имя метода интерфейса безопасности, который будет включен в EDL-файл.
- `OUTPUT_DIR` – директория, где будет создан EDL-файл. По умолчанию `${CMAKE_CURRENT_BINARY_DIR}`.

В результате работы команды переменная `EDL_FILE` экспортируется и содержит путь до сгенерированного EDL-файла.

Пример вызова:

```
generate_edl_file(${ENTITY_NAME} EDL_COMPONENTS "env: kl.Env")
```

Пример использования команды см. в статье "Файлы CMakeLists.txt для сборки прикладных программ".

nk_build_idl_files()

Команда объявлена в файле `/opt/KasperskyOS-Community-Edition-<version>toolchain/share/cmake/Modules/platform/nk2.cmake`.

```
nk_build_idl_files(NAME ...)
```

Команда создает `CMake`-цель для генерации `.idl.h`-файлов для одного или нескольких заданных IDL-файлов при помощи компилятора NK.

Параметры:

- `NAME` – имя `CMake`-цели для сборки `.idl.h`-файлов. Если цель еще не создана, то она будет создана с помощью `add_library()` с указанным именем. Обязательный параметр.
- `NOINSTALL` – если указана эта опция, то файлы будут только сгенерированы в рабочей директории, но не будут установлены в глобальные директории: `${CMAKE_BINARY_DIR}/_headers_ ${CMAKE_BINARY_DIR}/_headers_/${PROJECT_NAME}`.

- `NK_MODULE` – в этом параметре необходимо указать имя пакета, исключив из него имя IDL-файла. Например, если в IDL-описании имя пакета задано как `kl.core.NameServer`, то в параметре `NK_MODULE` необходимо передать значение `kl.core`.

- `WORKING_DIRECTORY` – рабочая директория для вызова компилятора NK, по умолчанию: `${CMAKE_CURRENT_BINARY_DIR}`.

- `DEPENDS` – дополнительные цели сборки, от которых зависит IDL-файл.

  Для добавления нескольких целей нужно использовать несколько параметров `DEPENDS`.

- `IDL` – путь к IDL-файлу, для которого генерируется idl.h-файл. Обязательный параметр.

  Для добавления нескольких IDL-файлов нужно использовать несколько параметров `IDL`.

  > Если один IDL-файл импортирует другой IDL-файл, то генерацию idl.h-файлов нужно производить в порядке, необходимом для соблюдения зависимостей (сначала самые вложенные).

- `NK_FLAGS` – дополнительные флаги для NK компилятора.

Пример вызова:

```
nk_build_idl_files (echo_idl_files NK_MODULE "echo" IDL "resources/Ping.idl")
```

Пример использования команды см. в статье "Файлы CMakeLists.txt для сборки прикладных программ".

nk_build_cdl_files()

Команда объявлена в файле `/opt/KasperskyOS-Community-Edition-<version>toolchain/share/cmake/Modules/platform/nk2.cmake`.

```
nk_build_cdl_files(NAME ...)
```

Команда создает `CMake`-цель для генерации `.cdl.h`-файлов для одного или нескольких заданных CDL-файлов при помощи компилятора NK.

Параметры:

- `NAME` – имя `CMake`-цели для сборки `.cdl.h`-файлов. Если цель еще не создана, то она будет создана с помощью `add_library()` с указанным именем. Обязательный параметр.

- `NOINSTALL` – если указана эта опция, то файлы будут только сгенерированы в рабочей директории, но не установлены в глобальные директории: `${CMAKE_BINARY_DIR}/_headers_ ${CMAKE_BINARY_DIR}/_headers_/${PROJECT_NAME}`.

- `IDL_TARGET` – цель сборки `.idl.h`-файлов для IDL-файлов, содержащих описания служб, предоставляемых компонентами, описанными в CDL-файлах.

- `NK_MODULE` – в этом параметре необходимо указать имя компонента, исключив из него имя CDL-файла. Например, если в CDL-описании имя компонента задано как `kl.core.NameServer`, то в параметре `NK_MODULE` необходимо передать значение `kl.core`.

- `WORKING_DIRECTORY` – рабочая директория для вызова компилятора NK, по умолчанию: `${CMAKE_CURRENT_BINARY_DIR}`.
- `DEPENDS` – дополнительные цели сборки, от которых зависит CDL-файл.

  Для добавления нескольких целей нужно использовать несколько параметров `DEPENDS`.
- `CDL` – путь к CDL-файлу, для которого генерируется `.cdl.h`-файл. Обязательный параметр.

  Для добавления нескольких CDL-файлов нужно использовать несколько параметров `CDL`.
- `NK_FLAGS` – дополнительные флаги для NK компилятора.

Пример вызова:

```
nk_build_cdl_files (echo_cdl_files IDL_TARGET echo_idl_files NK_MODULE "echo" CDL
"resources/Ping.cdl")
```

Пример использования команды см. в статье "Файлы CMakeLists.txt для сборки прикладных программ".

nk_build_edl_files()

Команда объявлена в файле `/opt/KasperskyOS-Community-Edition-<version>toolchain/share/cmake/Modules/platform/nk2.cmake`.

```
nk_build_edl_files(NAME ...)
```

Команда создает `CMake`-цель для генерации `.edl.h`-файла для одного заданного EDL-файла при помощи компилятора NK.

Параметры:

- `NAME` – имя `CMake`-цели сборки `.edl.h`-файла. Если цель еще не создана, то она будет создана с помощью `add_library()` с указанным именем. Обязательный параметр.
- `NOINSTALL` – если указана эта опция, то файлы будут только сгенерированы в рабочей директории, но не установлены в глобальные директории: `${CMAKE_BINARY_DIR}/_headers_ ${CMAKE_BINARY_DIR}/_headers_/${PROJECT_NAME}`.
- `CDL_TARGET` – цель сборки `.cdl.h`-файлов для CDL-файлов, содержащих описания компонентов EDL-файла, для которого выполняется сборка.
- `IDL_TARGET` – цель сборки .idl.h-файлов для IDL-файлов, содержащих описания интерфейсов EDL-файла, для которого выполняется сборка.
- `NK_MODULE` – в этом параметре необходимо указать имя класса процессов, исключив из него имя EDL-файла. Например, если в EDL-описании имя класса процессов задано как `kl.core.NameServer`, то в параметре `NK_MODULE` необходимо передать значение `kl.core`.
- `WORKING_DIRECTORY` – рабочая директория для вызова компилятора NK, по умолчанию: `${CMAKE_CURRENT_BINARY_DIR}`.
- `DEPENDS` – дополнительные цели сборки, от которых зависит EDL-файл.

Для добавления нескольких целей нужно использовать несколько параметров `DEPENDS`.

- `EDL` – путь к EDL файлу, для которого генерируется edl.h-файл. Обязательный параметр.
- `NK_FLAGS` – дополнительные флаги для NK компилятора.

Примеры вызова:

```
nk_build_edl_files (echo_server_edl_files CDL_TARGET echo_cdl_files NK_MODULE "echo"
EDL "resources/Server.edl")
nk_build_edl_files (echo_client_edl_files NK_MODULE "echo" EDL "resources/Client.edl")
```

Пример использования команды см. в статье "Файлы CMakeLists.txt для сборки прикладных программ".

Генерация транспортного кода для разработки на языке C++

Для генерации транспортных прокси-объектов и стабов с помощью генератора `nkppmeta` при сборке решения используются `CMake`-команды add_nk_idl(), add_nk_cdl() и add_nk_edl().

add_nk_idl()

Команда объявлена в файле `/opt/KasperskyOS-Community-Edition-<version>/toolchain/share/cmake/Modules/platform/nk2.cmake`.

```
add_nk_idl(NAME IDL_FILE ...)
```

Команда создает `CMake`-цель для генерации заголовочного файла `*.idl.cpp.h` для заданного IDL-файла при помощи компилятора `nkppmeta`. Также команда создает библиотеку, содержащую транспортный код для заданного интерфейса. Для компоновки с этой библиотекой необходимо использовать команду `bind_nk_targets()`.

Генерируемые заголовочные файлы содержат представление на языке C++ для интерфейса и типов данных, описанных в IDL-файле, а также методы, необходимые для использования прокси-объектов и стабов.

Параметры:

- `NAME` – имя `CMake`-цели. Обязательный параметр.
- `IDL_FILE` – путь к IDL-файлу. Обязательный параметр.
- `NK_MODULE` – в этом параметре необходимо указать имя пакета, исключив из него имя IDL-файла. Например, если в IDL-описании имя пакета задано как `kl.core.NameServer`, то в параметре `NK_MODULE` необходимо передать значение `kl.core`.
- `LANG` – в этом параметре необходимо указать значение `CXX`.

Пример вызова:

```
add_nk_idl (ANIMAL_IDL "${CMAKE_SOURCE_DIR}/resources/Animal.idl"
            NK_MODULE  "example"
            LANG       "CXX")
```

add_nk_cdl()

Команда объявлена в файле `/opt/KasperskyOS-Community-Edition-<version>/toolchain/share/cmake/Modules/platform/nk2.cmake`.

```
add_nk_cdl(NAME CDL_FILE ...)
```

Команда создает `CMake`-цель для генерации файла `*.cdl.cpp.h` для заданного CDL-файла при помощи компилятора `nkppmeta`. Команда также создает библиотеку, содержащую транспортный код для заданного компонента. Для компоновки с этой библиотекой необходимо использовать команду `bind_nk_targets()`.

Файл `*.cdl.cpp.h` содержит дерево вложенных компонентов и служб, предоставляемых компонентом, описанным в CDL-файле.

Параметры:

- `NAME` – имя `CMake`-цели. Обязательный параметр.
- `CDL_FILE` – путь к CDL-файлу. Обязательный параметр.
- `NK_MODULE` – в этом параметре необходимо указать имя компонента, исключив из него имя CDL-файла. Например, если в CDL-описании имя компонента задано как `kl.core.NameServer`, то в параметре `NK_MODULE` необходимо передать значение `kl.core`.
- `LANG` – в этом параметре необходимо указать значение `CXX`.

Пример вызова:

```
add_nk_cdl (CAT_CDL   "${CMAKE_SOURCE_DIR}/resources/Cat.cdl"
            NK_MODULE "example"
            LANG      "CXX")
```

add_nk_edl()

Команда объявлена в файле `/opt/KasperskyOS-Community-Edition-<version>/toolchain/share/cmake/Modules/platform/nk2.cmake`.

```
add_nk_edl(NAME EDL_FILE ...)
```

Команда создает `CMake`-цель для генерации файла `*.edl.cpp.h` для заданного EDL-файла при помощи компилятора `nkppmeta`. Также команда создает библиотеку, содержащую транспортный код для серверной или клиентской программы. Для компоновки с этой библиотекой необходимо использовать команду `bind_nk_targets()`.

Файл `*.edl.cpp.h` содержит дерево вложенных компонентов и служб, предоставляемых классом процессов, описанным в EDL-файле.

Параметры:

- `NAME` – имя `CMake`-цели. Обязательный параметр.
- `EDL_FILE` – путь к EDL-файлу. Обязательный параметр.
- `NK_MODULE` – в этом параметре необходимо указать имя класса процессов, исключив из него имя EDL-файла. Например, если в EDL-описании имя класса процессов задано как `kl.core.NameServer`, то в параметре `NK_MODULE` необходимо передать значение `kl.core`.
- `LANG` – в этом параметре необходимо указать значение `CXX`.

Пример вызова:

```
add_nk_edl (SERVER_EDL "${CMAKE_SOURCE_DIR}/resources/Server.edl"
            NK_MODULE  "example"
            LANG       "CXX")
```

Библиотека image

Этот раздел содержит описание команд и макросов `CMake`-библиотеки `image` , входящей в состав KasperskyOS Community Edition и содержащей скрипты сборки образа решения.

build_kos_qemu_image()

Команда объявлена в файле `/opt/KasperskyOS-Community-Edition-<version>toolchain/share/cmake/Modules/platform/image.cmake`.

```
build_kos_qemu_image(NAME ...)
```

Команда создает `CMake`-цель сборки образа решения для QEMU.

Параметры:

- `NAME` – имя `CMake`-цели для сборки образа решения. Обязательный параметр.
- `PERFCNT_KERNEL` – использовать ядро со счетчиками производительности, если оно доступно в составе KasperskyOS Community Edition.
- `EINIT_ENTITY` – имя исполняемого файла, из которого будет запускаться программа `Einit`.

- `EXTRA_XDL_DIR` – дополнительные директории для включения при сборке программы `Einit`.
- `CONNECTIONS_CFG` – путь до файла `init.yaml` или шаблона init.yaml.in.
- `SECURITY_PSL` – путь до файла `security.psl` или шаблона security.psl.in.
- `KLOG_ENTITY` – цель сборки системной программы `Klog`, отвечающей за аудит безопасности. Если цель не указана – аудит не выполняется.
- `QEMU_FLAGS` – дополнительные флаги для запуска QEMU.
- `IMAGE_BINARY_DIR_BIN` – директория для финального образа и других артефактов, по умолчанию совпадает с `CMAKE_CURRENT_BINARY_DIR`.
- `NO_AUTO_BLOB_CONTAINER` – не включать в образ решения программу `BlobContainer`, необходимую для работы с динамическими библиотеками в разделяемой памяти. Подробнее см. "Включение системной программы BlobContainer в решение на базе KasperskyOS".
- `PACK_DEPS`, `PACK_DEPS_COPY_ONLY`, `PACK_DEPS_LIBS_PATH`, `PACK_DEPS_COPY_TARGET` – параметры, задающие способ добавления динамических библиотек в образ решения.
- `IMAGE_FILES` – исполняемые файлы прикладных и системных программ (кроме программы `Einit`) и любые другие файлы для добавления в образ ROMFS.

  Для добавления нескольких программ или файлов можно использовать несколько параметров `IMAGE_FILES`.
- `<пути до файлов>` – свободные параметры, тоже что `IMAGE_FILES`.

Пример вызова:

```
build_kos_qemu_image (   kos-qemu-image
                         EINIT_ENTITY EinitQemu
                         CONNECTIONS_CFG "src/init.yaml.in"
                         SECURITY_CFG "src/security.cfg.in"
                         IMAGE_FILES ${ENTITIES})
```

Пример использования команды см. в статье "Файлы CMakeLists.txt для сборки программы Einit".

build_kos_hw_image()

Команда объявлена в файле `/opt/KasperskyOS-Community-Edition-<version>toolchain/share/cmake/Modules/platform/image.cmake`.

```
build_kos_hw_image(NAME ...)
```

Команда создает `CMake`-цель сборки образа решения для аппаратной платформы.

Параметры:

- `NAME` – имя `CMake`-цели для сборки образа решения. Обязательный параметр.

- `PERFCNT_KERNEL` – использовать ядро со счетчиками производительности, если оно доступно в составе KasperskyOS Community Edition.
- `EINIT_ENTITY` – имя исполняемого файла, из которого будет запускаться программа `Einit`.
- `EXTRA_XDL_DIR` – дополнительные директории для включения при сборке программы `Einit`.
- `CONNECTIONS_CFG` – путь до файла init.yaml или шаблона init.yaml.in.
- `SECURITY_PSL` – путь до файла security.psl или шаблона security.psl.in.
- `KLOG_ENTITY` – цель сборки системной программы `Klog`, отвечающей за аудит безопасности. Если цель не указана – аудит не выполняется.
- `IMAGE_BINARY_DIR_BIN` – директория для финального образа и других артефактов, по умолчанию `CMAKE_CURRENT_BINARY_DIR`.
- `NO_AUTO_BLOB_CONTAINER` – не включать в образ решения программу `BlobContainer`, необходимую для работы с динамическими библиотеками в разделяемой памяти. Подробнее см. "Включение системной программы BlobContainer в решение на базе KasperskyOS".
- `PACK_DEPS`, `PACK_DEPS_COPY_ONLY`, `PACK_DEPS_LIBS_PATH`, `PACK_DEPS_COPY_TARGET` – параметры, задающие способ добавления динамических библиотек в образ решения.
- `IMAGE_FILES` – исполняемые файлы прикладных и системных программ (кроме программы `Einit`) и любые другие файлы для добавления в образ ROMFS.

  Для добавления нескольких программ или файлов можно использовать несколько параметров `IMAGE_FILES`.
- `<пути до файлов>` – свободные параметры, тоже что `IMAGE_FILES`.

Пример вызова:

```
build_kos_hw_image ( kos-image
                     EINIT_ENTITY EinitHw
                     CONNECTIONS_CFG "src/init.yaml.in"
                     SECURITY_CFG "src/security.cfg.in"
                     IMAGE_FILES ${ENTITIES})
```

Пример использования команды см. в статье "Файлы CMakeLists.txt для сборки программы Einit".

Сборка без использования CMake

Этот раздел содержит описание скриптов, утилит, компиляторов и шаблонов сборки, поставляемых в KasperskyOS Community Edition.

Эти инструменты можно использовать:

- в других системах сборки;
- для выполнения отдельных шагов сборки;
- для изучения особенностей сборки и написания собственной системы сборки.

Общая схема сборки образа решения приведена в статье "Общая схема сборки".

Инструменты для сборки решения

Этот раздел содержит описание скриптов, утилит, компиляторов и шаблонов сборки, поставляемых в KasperskyOS Community Edition.

Утилиты и скрипты сборки

В состав KasperskyOS Community Edition входят следующие утилиты и скрипты сборки:

- nk-gen-c

  Компилятор NK (`nk-gen-c`) генерирует транспортный код на основе IDL-, CDL- и EDL-описаний. Транспортный код нужен для формирования, отправки, приема и обработки IPC-сообщений.

- nk-psl-gen-c

  Компилятор `nk-psl-gen-c` генерирует исходный код модуля безопасности Kaspersky Security Module на языке C на основе описания политики безопасности решения и IDL-, CDL-, EDL-описаний. Также компилятор `nk-psl-gen-c` позволяет генерировать исходный код тестов политики безопасности решения на языке C на основе тестов политики безопасности решения на языке PAL.

- einit

  Утилита `einit` позволяет автоматизировать создание кода инициализирующей программы `Einit`. Эта программа первой запускается при загрузке KasperskyOS и запускает остальные программы, а также создает IPC-каналы между ними.

- makekss

  Скрипт `makekss` создает модуль безопасности Kaspersky Security Module.

- makeimg

  Скрипт `makeimg` создает финальный загружаемый образ решения на базе KasperskyOS со всеми запускаемыми программами и модулем Kaspersky Security Module.

nk-gen-c

Компилятор NK (`nk-gen-c`) генерирует транспортный код на основе IDL-, CDL-, EDL-описаний.

Компилятор `nk-gen-c` принимает IDL-, CDL- или EDL-файл и создает следующие файлы:

- Файл `*.*dl.h`, содержащий транспортный код.
- Файл `*.*dl.nk.d`, в котором перечислены зависимости созданного файла `*.*dl.h` от IDL- и CDL-файлов. Файл `*.*dl.nk.d` создается для системы сборки.

Синтаксис shell-команды для запуска компилятора `nk-gen-c`:

```
nk-gen-c [-I <PATH>]... [-o <PATH>] [--types] [--interface] [--endpoints]
[--client] [--server] [--extended-errors] [--trace-client-ipc {headers|dump}]
[--trace-server-ipc {headers|dump}] [--ipc-trace-method-filter <METHOD>[,METHOD]...]
[-h|--help] [--version] <FILE>
```

Базовые параметры:

- `FILE`

  Путь к IDL-, CDL- или EDL-файлу, для которого необходимо сгенерировать транспортный код.

- `-I <PATH>`

  Через эти параметры задаются пути к директориям, которые содержат вспомогательные файлы, необходимые для генерации транспортного кода. (Вспомогательные файлы располагаются в директории `sysroot-*-kos/include` из состава KasperskyOS SDK.) Также через эти параметры можно задать пути к директориям, содержащим IDL-, CDL-файлы, на которые ссылается файл, заданный через параметр `FILE`.

- `-o <PATH>`

  Путь к существующей директории, в которую будут помещены созданные файлы. Если этот параметр не указан, созданные файлы будут помещены в текущую директорию.

- `-h|--help`

  Выводит текст справки.

- `--version`

  Выводит версию компилятора `nk-gen-c`.

- `--extended-errors`

  Это параметр обеспечивает возможность использовать интерфейсные методы с одним или несколькими error-параметрами произвольных IDL-типов. (Клиент работает с error-параметрами как с выходными параметрами.)

  > Если не указывать параметр `--extended-errors`, можно использовать интерфейсные методы только с одним error-параметром `status` IDL-типа `UInt16`, значение которого передается клиенту через код возврата интерфейсного метода. Такой способ является устаревшим и перестанет поддерживаться в будущем, поэтому рекомендуется всегда указывать параметр `--extended-errors`.

Выборочная генерация транспортного кода

Чтобы уменьшить объем генерируемого транспортного кода, можно использовать флаги выборочной генерации транспортного кода. Например, для программ, реализующих службы, можно использовать флаг `--server`, а для программ, использующих службы, можно использовать флаг `--client`.

> Если ни один из флагов выборочной генерации транспортного кода не указан, компилятор `nk-gen-c` генерирует для заданного IDL-, CDL- или EDL-файла транспортный код со всеми возможными методами и типами.

Флаги выборочной генерации транспортного кода для IDL-файла:

- `--types`

Транспортный код включает типы, соответствующие IDL-типам из заданного IDL-файла, а также импортируемым в этот файл IDL-типам, которые используются в IDL-типах заданного IDL-файла. При этом типы, соответствующие импортируемым IDL-константам и псевдонимам импортируемых IDL-типов, не включаются в файл `*.idl.h`. Чтобы использовать типы, соответствующие импортируемым IDL-константам и псевдонимам импортируемых IDL-типов, нужно отдельно сгенерировать транспортный код для IDL-файлов, из которых осуществляется импорт.

- `--interface`

  Транспортный код соответствует флагу `--types`, а также включает типы структур фиксированной части IPC-запросов и IPC-ответов для интерфейсных методов, сигнатуры которых указаны в заданном IDL-файле. Кроме того, транспортный код содержит константы с размерами арен IPC-сообщений.

- `--client`

  Транспортный код соответствует флагу `--interface`, а также включает тип прокси-объекта, метод инициализации прокси-объекта и интерфейсные методы, указанные в заданном IDL-файле.

- `--server`

  Транспортный код соответствует флагу `--interface`, а также включает типы и диспетчер (dispatch-метод), используемые для обработки IPC-запросов, соответствующих интерфейсным методам, указанным в заданном IDL-файле.

Флаги выборочной генерации транспортного кода для CDL- или EDL-файла:

- `--types`

  Транспортный код включает типы, соответствующие IDL-типам, которые используются в параметрах методов служб, предоставляемых компонентом (для заданного CDL-файла) или классом процессов (для заданного EDL-файла).

- `--endpoints`

  Транспортный код соответствует флагу `--types`, а также включает типы структур фиксированной части IPC-запросов и IPC-ответов для методов служб, предоставляемых компонентом (для заданного CDL-файла) или классом процессов (для заданного EDL-файла). Кроме того, транспортный код содержит константы с размерами арен IPC-сообщений.

- `--client`

  Транспортный код соответствует флагу `--types`, а также включает типы структур фиксированной части IPC-запросов и IPC-ответов для методов служб, предоставляемых компонентом (для заданного CDL-файла) или классом процессов (для заданного EDL-файла). Кроме того, транспортный код содержит константы с размерами арен IPC-сообщений, а также типы прокси-объектов, методы инициализации прокси-объектов и методы служб, предоставляемых компонентом (для заданного CDL-файла) или классом процессов (для заданного EDL-файла).

- `--server`

  Транспортный код соответствует флагу `--types`, а также включает типы и диспетчеры (dispatch-методы), используемые для обработки IPC-запросов, соответствующих службам, предоставляемым компонентом (для заданного CDL-файла) или классом процессов (для заданного EDL-файла). Кроме того, транспортный код содержит константы с размерами арен IPC-сообщений, а также типы стабов, методы инициализации стабов и типы структур фиксированной части IPC-запросов и IPC-ответов для методов служб, предоставляемых компонентом (для заданного CDL-файла) или классом процессов (для заданного EDL-файла).

Вывод диагностических данных об отправке и приеме IPC-сообщений

Транспортный код может формировать диагностические данные об отправке и приеме IPC-сообщений и выводить эти данные через стандартный вывод ошибок. Чтобы генерировать транспортный код с такими возможностями, нужно использовать следующие параметры:

- `--trace-client-ipc {headers|dump}`

  Код вывода диагностических данных исполняется непосредственно перед выполнением системного вызова `Call()` и сразу после его выполнения. Если указано значение `headers`, диагностические данные включают идентификатор метода службы (MID), идентификатор службы (RIID), размер фиксированной части IPC-сообщения в байтах, содержимое дескриптора арены IPC-сообщения, размер арены IPC-сообщения в байтах и размер использованной части арены IPC-сообщения в байтах. Если указано значение `dump`, диагностические данные дополнительно включают содержимое фиксированной части и арены IPC-сообщения в шестнадцатеричном представлении.

  При использовании этого параметра нужно указать флаг выборочной генерации транспортного кода `--client` либо не указывать флаги выборочной генерации транспортного кода.

- `--trace-server-ipc {headers|dump}`

  Код вывода диагностических данных исполняется непосредственно перед вызовом функции, реализующей интерфейсный метод, и сразу после выполнения этой функции, то есть при вызове диспетчера (dispatch-метода) в промежутке между выполнением системных вызовов `Recv()` и `Reply()`. Если указано значение `headers`, диагностические данные включают идентификатор метода службы (MID), идентификатор службы (RIID), размер фиксированной части IPC-сообщения в байтах, содержимое дескриптора арены IPC-сообщения, размер арены IPC-сообщения в байтах и размер использованной части арены IPC-сообщения в байтах. Если указано значение `dump`, диагностические данные дополнительно включают содержимое фиксированной части и арены IPC-сообщения в шестнадцатеричном представлении.

  При использовании этого параметра нужно указать флаг выборочной генерации транспортного кода `--server` либо не указывать флаги выборочной генерации транспортного кода.

- `--ipc-trace-method-filter <METHOD>[,METHOD]...`

  Вывод диагностических данных выполняется, если только вызваны заданные интерфейсные методы. В качестве значения `METHOD` можно использовать имя интерфейсного метода либо конструкцию `<имя пакета>:<имя интерфейсного метода>`. Имя пакета и имя интерфейсного метода указаны в IDL-файле.

  Если этот параметр не указан, вывод диагностических данных выполняется при вызове любого интерфейсного метода.

  Параметр можно указать многократно. Например, можно указать все требуемые интерфейсные методы в одном параметре или каждый требуемый интерфейсные метод в отдельном параметре.

nk-psl-gen-c

Компилятор `nk-psl-gen-c` генерирует исходный код модуля безопасности Kaspersky Security Module на языке C на основе описания политики безопасности решения и IDL-, CDL-, EDL-описаний. Этот код используется скриптом makekss.

Компилятор `nk-psl-gen-c` также позволяет генерировать исходный код тестов политики безопасности решения на языке C на основе тестов политики безопасности решения на языке PAL.

Синтаксис shell-команды для запуска компилятора `nk-psl-gen-c`:

```
nk-psl-gen-c [{-I|--include-dir} <DIR>]... [{-o|--output} <FILE>] [--out-tests <FILE>]
[{-t|--tests} <ARG>] [{-a|--audit} <FILE>] [-h|--help] [--version] <INPUT>
```

Параметры:

- `INPUT`

  Путь к верхнеуровневому файлу описания политики безопасности решения. Как правило, это файл `security.psl`.

- `{-I|--include-dir} <DIR>`

  Через эти параметры задаются пути к директориям с IDL-, CDL-, EDL-файлам, относящимися к решению, и пути к директориям, которые содержат вспомогательные файлы из состава KasperskyOS SDK (`common`, `sysroot-*-kos/include`, `toolchain/include`).

- `{-o|--output} <FILE>`

  Путь к файлу, в который будет сохранен исходный код модуля безопасности Kaspersky Security Module и опционально исходный код тестов политики безопасности решения. Путь должен включать существующие директории.

- `--out-tests <FILE>`

  Путь к файлу, в который будет сохранен исходный код тестов политики безопасности решения.

- `{-t|--tests} <ARG>`

  Задает, нужно ли генерировать исходный код тестов политики безопасности решения. `ARG` может принимать следующие значения:

  - `skip` – исходный код тестов не генерируется. Это значение используется по умолчанию, если параметр `-t, --tests <ARG>` не указан.

  - `generate` – исходный код тестов генерируется. Если исходный код тестов генерируется, то рекомендуется использовать параметр `--out-tests <FILE>`, иначе исходный код тестов будет сохранен в одном файле с исходным кодом модуля безопасности Kaspersky Security Module, что может привести к ошибкам при сборке.

- `{-a|--audit} <FILE>`

  Путь к файлу, в который будет сохранен исходный код декодера аудита на языке C.

- `-h|--help`

  Выводит текст справки.

- `--version`

  Выводит версию компилятора `nk-psl-gen-c`.

einit

Утилита `einit` позволяет автоматизировать создание кода инициализирующей программы Einit.

Утилита `einit` принимает описание инициализации решения (по умолчанию файл `init.yaml`), а также EDL-, CDL- и IDL-описания, и создает файл с исходным кодом инициализирующей программы `Einit`. Программу `Einit` затем необходимо собрать с помощью кросс-компилятора языка C, поставляемого в KasperskyOS Community Edition.

Синтаксис использования утилиты `einit`:

```
einit -I PATH -o PATH [--help] FILE
```

Параметры:

- `FILE`

  Путь к файлу `init.yaml`.

- `-I PATH`

  Путь к директории, содержащей вспомогательные файлы (включая EDL-, CDL- и IDL-описания), необходимые для генерации инициализирующей программы. По умолчанию эти файлы располагаются в директории `/opt/KasperskyOS-Community-Edition-<version>/sysroot-aarch64-kos/include`.

- `-o, --out-file PATH`

  Путь к создаваемому .c файлу с кодом инициализирующей программы.

- `-h, --help`

  Отображает текст справки.

makekss

Скрипт `makekss` создает модуль безопасности Kaspersky Security Module.

Скрипт вызывает компилятор nk-psl-gen-c для генерации исходного кода модуля безопасности и затем компилирует полученный код, вызывая компилятор C, поставляемый в KasperskyOS Community Edition.

Скрипт создает модуль безопасности из описания политики безопасности решения.

Синтаксис использования скрипта `makekss`:

```
makekss --target=ARCH --module=PATH --with-nk="PATH" --with-nktype="TYPE" --with-
nkflags="FLAGS" [--output="PATH"][--help][--with-cc="PATH"][--with-cflags="FLAGS"]
FILE
```

Параметры:

- `FILE`

  Путь к верхнеуровневому файлу описания политики безопасности решения.

- `--target=ARCH`

  Архитектура процессора, для которой производится сборка.

- `--module=-lPATH`

  Путь к библиотеке `ksm_kss`. Этот ключ передается компилятору C для компоновки с этой библиотекой.

- `--with-nk=PATH`

  Путь к компилятору `nk-psl-gen-c`, который будет использоваться для генерации исходного кода модуля безопасности. По умолчанию компилятор расположен в `/opt/KasperskyOS-Community-Edition-<version>/toolchain/bin/nk-psl-gen-c`.

- `--with-nktype="TYPE"`

  Указывает на тип компилятора NK, который будет использоваться. Для использования компилятора `nk-psl-gen-c`, необходимо указать тип `psl`.

- `--with-nkflags="FLAGS"`

  Параметры, с которыми вызывается компилятор `nk-psl-gen-c`.

  Компилятору `nk-psl-gen-c` потребуется доступ ко всем EDL- CDL- и IDL-описаниям. Для того, чтобы компилятор `nk-psl-gen-c` мог найти эти описания, нужно передать пути к расположению этих описаний в параметре `--with-nkflags`, используя параметр `-I` компилятора `nk-psl-gen-c`.

- `--output=PATH`

  Путь к создаваемому файлу модуля безопасности.

- `--with-cc=PATH`

  Путь к компилятору C, который будет использоваться для сборки модуля безопасности. По умолчанию используется компилятор, поставляемый в KasperskyOS Community Edition.

- `--with-cflags=FLAGS`

  Параметры, с которыми вызывается компилятор C.

- `-h, --help`

  Отображает текст справки.

makeimg

Скрипт `makeimg` создает финальный загружаемый образ решения на базе KasperskyOS со всеми исполняемыми файлами программ и модулем Kaspersky Security Module.

Скрипт принимает список файлов, включая исполняемые файлы всех программ, которые нужно добавить в ROMFS загружаемого образа, и создает следующие файлы:

- образ решения;
- образ решения без таблиц символов (`.stripped`);
- образ решения с отладочными таблицами символов (`.dbg.syms`).

Синтаксис использования скрипта makeimg:

```
makeimg --target=ARCH --sys-root=PATH --with-toolchain=PATH --ldscript=PATH --img-
src=PATH --img-dst=PATH --with-init=PATH [--with-extra-asflags=FLAGS][--with-extra-
ldflags=FLAGS][--help] FILES
```

Параметры:

- `FILES`

  Список путей к файлам, включая исполняемые файлы всех программ, которые нужно добавить в ROMFS.

  Модуль безопасности (`ksm.module`) нужно указывать явно, иначе он не будет включен в образ решения. Программу `Einit` указывать не нужно, так как она будет включена в образ решения автоматически.

- `--target=ARCH`

  Архитектура, для которой производится сборка.

- `--sys-root=PATH`

  Путь к корневой директории sysroot. По умолчанию эта директория расположена в `/opt/KasperskyOS-Community-Edition-<version>/sysroot-aarch64-kos/`.

- `--with-toolchain=PATH`

  Путь к набору вспомогательных утилит, необходимых для сборки решения. По умолчанию эти утилиты расположены в `/opt/KasperskyOS-Community-Edition-<version>/toolchain/`.

- `--ldscript=PATH`

  Путь к скрипту компоновщика, необходимому для сборки решения. По умолчанию этот скрипт расположен в `/opt/KasperskyOS-Community-Edition-<version>/libexec/aarch64-kos/`.

- `--img-src=PATH`

  Путь к заранее скомпилированному ядру KasperskyOS. По умолчанию ядро расположено в `/opt/KasperskyOS-Community-Edition-<version>/libexec/aarch64-kos/`.

- `--img-dst=PATH`

  Путь к создаваемому файлу образа.

- `--with-init=PATH`

  Путь к исполняемому файлу инициализирующей программы `Einit`.

- `--with-extra-asflags=FLAGS`

  Дополнительные флаги для ассемблера AS.

- `--with-extra-ldflags=FLAGS`

  Дополнительные флаги для компоновщика LD.

- `-h, --help`

  Отображает текст справки.

Кросс-компиляторы

В тулчейн из состава KasperskyOS SDK входит один или несколько компиляторов GCC. В директории `toolchain/bin` находятся следующие файлы:

- исполняемые файлы компиляторов (например, `x86_64-pc-kos-gcc`, `arm-kos-g++`);
- исполняемые файлы компоновщиков (например, `x86_64-pc-kos-ld`, `arm-kos-ld`);
- исполняемые файлы ассемблеров (например, `x86_64-pc-kos-as`, `arm-kos-as`);

В GCC, кроме стандартных макросов, определен дополнительный макрос `__KOS__=1`. Использование этого макроса упрощает портирование программного кода на KasperskyOS, а также разработку платформонезависимых программ.

Чтобы просмотреть список стандартных макросов GCC, выполните следующую команду:

```
echo '' | aarch64-kos-gcc -dM -E -
```

Особенности работы компоновщика

При выполнении сборки исполняемого файла программы компоновщик по умолчанию связывает следующие библиотеки в указанном порядке:

1. libc – стандартная библиотека языка C.
2. libm – библиотека, реализующая математические функции стандартной библиотеки языка C.
3. libvfs_stubs – библиотека, содержащая заглушки функций ввода/вывода (например, `open`, `socket`, `read`, `write`).
4. libkos – библиотека для доступа к службам ядра KasperskyOS.
5. libenv – библиотека подсистемы настройки окружения программ (переменных окружения, аргументов функции `main` и пользовательских конфигураций).
6. libsrvtransport-u – библиотека поддержки IPC между процессами и ядром.

Пример сборки без использования CMake

Ниже приведен пример скрипта для сборки простейшего примера. Этот пример содержит единственную прикладную программу `Hello`, которая не предоставляет службы.

> Приведенный скрипт предназначен только для демонстрации используемых команд сборки.

build.sh

```
#!/bin/sh

# В переменной SDK нужно указать путь к директории установки KasperskyOS Community
Edition.
SDK=/opt/KasperskyOS-Community-Edition-<version>
TOOLCHAIN=$SDK/toolchain
SYSROOT=$SDK/sysroot-aarch64-kos

PATH=$TOOLCHAIN/bin:$PATH

# Создание файла Hello.edl.h из Hello.edl
# (Программа Hello не реализует служб, поэтому cdl- и idl-файлы отсутствуют.)
nk-gen-c -I $SYSROOT/include Hello.edl

# Компиляция и сборка программы Hello
aarch64-kos-gcc -o hello hello.c

# Создание модуля безопасности Kaspersky Security Module (ksm.module)
makekss --target=aarch64-kos                                  \
        --module=-lksm_kss                                        \
        --with-nkflags="-I $SDK/examples/common -I $SYSROOT/include" \
```

```
        security.psl

# Создание кода инициализирующей программы Einit
einit -I $SYSROOT/include -I . init.yaml -o einit.c

# Компиляция и сборка программы Einit
aarch64-kos-gcc -I . -o einit einit.c

# Создание загружаемого образа решения (kos-qemu-image)
makeimg --target=aarch64-kos                                \
        --sys-root=$SYSROOT                                   \
        --with-toolchain=$TOOLCHAIN                           \
        --ldscript=$SDK/libexec/aarch64-kos/kos-qemu.ld   \
        --img-src=$SDK/libexec/aarch64-kos/kos-qemu       \
        --img-dst=kos-qemu-image                              \
        Hello ksm.module

# Запуск решения под QEMU
qemu-system-aarch64 -m 1024 -serial stdio -kernel kos-qemu-image
```

Использование динамических библиотек

В решении на базе KasperskyOS можно использовать динамические библиотеки (файлы `*.so`). По сравнению со статическими библиотеками (файлами `*.a`) динамические библиотеки дают следующие преимущества:

- Экономия оперативной памяти.

  Несколько процессов могут использовать один экземпляр динамической библиотеки. Также программа и динамические библиотеки в одном процессе могут использовать один экземпляр динамической библиотеки.

  Динамические библиотеки могут загружаться в память и выгружаться из нее по инициативе программ, которые их используют.

- Удобство обновления ПО.

  Обновление динамической библиотеки распространяется на все зависимые от нее программы и динамические библиотеки без их повторной сборки.

- Возможность реализовать механизм плагинов.

  Плагины для компонентов решения представляют собой динамические библиотеки.

- Совместное использование кода и данных.

  Один экземпляр динамической библиотеки может совместно использоваться несколькими процессами, а также программой и динамическими библиотеками в одном процессе. Это позволяет, например, централизованно управлять множественным доступом к ресурсам или хранить общие данные.

> Динамические библиотеки поставляются в составе KasperskyOS SDK, а также могут быть созданы разработчиком решения на базе KasperskyOS. Работоспособность сторонних динамических библиотек не гарантируется.
> В настоящее время из-за технических ограничений в решении на базе KasperskyOS нельзя использовать `libc.so` и `libpthread.so`.

Условия, необходимые для использования динамических библиотек

Чтобы использовать динамические библиотеки в решении на базе KasperskyOS, нужно выполнить следующие условия:

1. Процессы, использующие динамические библиотеки, должны иметь доступ к файловым системам, в которых хранятся файлы динамических библиотек. Доступ к файловым системам обеспечивается VFS, которая может исполняться в контексте процессов, использующих динамические библиотеки, а также может быть отдельным процессом. VFS и другое ПО, с помощью которого VFS работает с накопителем (например драйвер накопителя), не должны использовать динамические библиотеки.

2. Тулчейн должен поддерживать динамическую компоновку.

   В составе KasperskyOS SDK поставляется отдельный тулчейн для каждой поддерживаемой архитектуры процессора. Требуемый тулчейн может не поддерживать динамическую компоновку. Чтобы проверить, что динамическая компоновка поддерживается, нужно использовать в корневом файле `CMakeLists.txt` `CMake`-команду `get_property()` следующим образом:

   ```
   get_property(CAN_SHARED GLOBAL PROPERTY TARGET_SUPPORTS_SHARED_LIBS)
   if(CAN_SHARED)
       message(STATUS "Dynamic linking is supported.")
   endif()
   ```

3. Исполняемый код программ, использующих динамические библиотеки, должен быть собран с флагом `-rdynamic` (с динамической компоновкой).

   Если тулчейн поддерживает динамическую компоновку, то `CMake`-команда `initialize_platform()` делает так, что для сборки всех исполняемых файлов, заданных через `CMake`-команды `add_executable()`, этот флаг используется автоматически.

> Если `CMake`-команда `initialize_platform(FORCE_STATIC)` вызывается в корневом файле `CMakeLists.txt`, то тулчейн, поддерживающий динамическую компоновку, выполняет статическую компоновку исполняемых файлов.
> `CMake`-команда `project_static_executable_header_default()` влияет на сборку исполняемых файлов, заданных через последующие `CMake`-команды `add_executable()` в одном файле `CMakeLists.txt`. Тулчейн, поддерживающий динамическую компоновку, выполняет статическую компоновку этих исполняемых файлов.
> `CMake`-команда `platform_target_force_static()` влияет на сборку одного исполняемого файла, заданного через `CMake`-команду `add_executable()`. Тулчейн, поддерживающий динамическую компоновку, выполняет статическую компоновку этого исполняемого файла.
> Исполняемый код программ, который собирается с флагом `-rdynamic`, компонуется со статической библиотекой, если динамическая библиотека не найдена. Например, если используется `CMake`-команда `target_link_libraries(client -lm)`, программа `client` компонуется со статической библиотекой `libm.a`, если динамическая библиотека `libm.so` не найдена.

Жизненный цикл динамической библиотеки

Жизненный цикл динамической библиотеки включает следующие стадии:

1. Загрузка в память.

Динамическая библиотека, скомпонованная с программой, загружается в память при запуске процесса, в контексте которого эта программа исполняется. Запущенный процесс может загрузить динамическую библиотеку в память вызовом функции `dlopen()` интерфейса POSIX. Динамическая библиотека может быть скомпонована с другими динамическими библиотеками, поэтому программа зависит не только от непосредственно скомпонованной с ней динамической библиотеки, но и от всего графа зависимостей этой библиотеки. Динамическая библиотека загружается в память совместно с всеми динамическими библиотеками, от которых зависит.

Если системная программа `BlobContainer` включена в решение на базе KasperskyOS, то один экземпляр динамической библиотеки загружается в разделяемую память независимо от того, сколько процессов использует эту библиотеку. (Точнее, в разделяемую память загружается только часть динамической библиотеки, включающая код и доступные только на чтение данные. Другая часть динамической библиотеки в любом случае загружается в память каждого процесса, который использует эту библиотеку.) Если системная программа `BlobContainer` не включена в решение, то отдельные экземпляры динамической библиотеки загружаются в память процессов, использующих эту библиотеку. Динамическая библиотека, от которой зависит нескольких других динамических библиотек, загружается в разделяемую память или память процесса в единственном экземпляре.

Если задать через переменную окружения `LD_PRELOAD` список динамических библиотек, то эти динамические библиотеки будут загружены в память, даже если программа не зависит от них. (Элементами списка должны быть абсолютные или относительные пути к динамическим библиотекам, разделенные двоеточием, например: `LD_PRELOAD=libmalloc.so:libfree.so:/usr/somepath/lib/libfoo.so`.) Функции, которые экспортируются динамическими библиотеками, указанными в `LD_PRELOAD`, замещают одноименные функции, которые экспортируются другими загруженными в разделяемую память или память процесса динамическими библиотеками. Это можно использовать для целей отладки, если требуется подменить функции, импортируемые из динамических библиотек.

Загрузчик динамических библиотек выполняет поиск динамических библиотек, от которых зависят программы, в следующем порядке:

1. По абсолютным путям, заданным через переменную окружения `LD_LIBRARY_PATH`.

   Пути должны быть разделены двоеточием, например: `LD_LIBRARY_PATH=/usr/lib:/home/user/lib`.

2. По абсолютным путям, заданным в поле `DT_RUNPATH` или `DT_RPATH` секции `.dynamic` исполняемых файлов и динамических библиотек.

   При компоновке исполняемых файлов и динамических библиотек могут быть заданы пути, по которым загрузчик динамических библиотек будет выполнять поиск. (Это можно сделать, например, через свойство `INSTALL_RPATH` в `CMake`-команде `set_target_properties()`.) Пути для поиска динамических библиотек сохраняются в поле `DT_RUNPATH` или `DT_RPATH` секции `.dynamic`. Это поле может быть как в исполняемых файлах, скомпонованных с динамическими библиотеки, так и в самих динамических библиотеках, скомпонованных с другими динамическими библиотеками.

3. По пути `/lib`.

Загрузчик динамических библиотек выполняет поиск в том же порядке, если в параметре `filename` функции `dlopen()` или в переменной окружения `LD_PRELOAD` указан относительный путь к динамической библиотеке. Если указан абсолютный путь, то загрузчик помещает динамическую библиотеку в память без выполнения поиска.

2. Использование процессом (процессами).

3. Выгрузка из памяти.

Динамическая библиотека выгружается из разделяемой памяти, когда все процессы, использующие эту библиотеку, завершились или вызвали функцию `dlclose()` интерфейса POSIX. Динамическая библиотека, загруженная в память процесса вызовом функции `dlopen()`, выгружается вызовом функции `dlclose()`. Динамическая библиотека, скомпонованная с программой, не может быть выгружена из памяти до завершения процесса, в контексте которого эта программа исполняется. Динамическая библиотека, скомпонованная с другими динамическими библиотеками, выгружается из памяти после выгрузки всех библиотек, которые зависят от нее, либо после завершения процесса.

Включение системной программы BlobContainer в решение на базе KasperskyOS

> Если программа `BlobContainer` поставляется в составе KasperskyOS SDK, ее необходимо включить в решение, в котором используются динамические библиотеки. Чтобы проверить, поставляется ли программа `BlobContainer` в составе KasperskyOS SDK, нужно убедиться в наличии исполняемого файла `sysroot-*-kos/bin/BlobContainer`.

Программа `BlobContainer` может быть включена в решение автоматически или вручную. Автоматическое включение этой программы в решение выполняется `CMake`-командами `build_kos_qemu_image()` и `build_kos_hw_image()`, если как минимум одна программа в решении скомпонована с динамической библиотекой. (Чтобы отключить автоматическое включение программы `BlobContainer` в решение, нужно добавить значение `NO_AUTO_BLOB_CONTAINER` в параметры `CMake`-команд `build_kos_qemu_image()` и `build_kos_hw_image()`.) Если программы в решении работают с динамическими библиотеками, используя только интерфейс POSIX (функции `dlopen()`, `dlsym()`, `dlerror()`, `dlclose()`), то программу `BlobContainer` нужно включить в решение вручную.

В случае применения программы `BlobContainer` должны быть созданы IPC-каналы от процессов, использующих динамические библиотеки, к процессу программы `BlobContainer`. Эти IPC-каналы могут быть созданы как статически, так и динамически. Если статически созданный IPC-канал отсутствует, клиентская и серверная части программы `BlobContainer` пытаются создать IPC-канал динамически, используя сервер имен.

Если программа `BlobContainer` включена в решение автоматически, то макросы `@INIT_EXTERNAL_ENTITIES@`, `@INIT_<имя программы>_ENTITY_CONNECTIONS@` и `@INIT_<имя программы>_ENTITY_CONNECTIONS+@`, используемые в файле `init.yaml.in`, автоматически создают в init-описании словари IPC-каналов, которые обеспечивают статическое создание IPC-каналов от процессов программ, скомпонованных с динамическими библиотеками, к процессу программы `BlobContainer`. (Процесс программы `BlobContainer` получает имя `kl.bc.BlobContainer`, а IPC-каналы получают имя `kl.BlobContainer`.) При этом для процессов, которые работают с динамическими библиотеками, используя только интерфейс POSIX, словари IPC-каналов к процессу программы `BlobContainer` автоматически не создаются, и, чтобы требуемые IPC-каналы были созданы статически, нужно создать эти словари вручную (эти IPC-каналы должны иметь имя `kl.BlobContainer`).

Если программа `BlobContainer` включена в решение вручную, и требуется статически создать IPC-каналы от процессов, использующих динамические библиотеки, к процессу программы `BlobContainer`, то нужно вручную создать словари необходимых IPC-каналов в init-описании. По умолчанию IPC-канал к процессу программы `BlobContainer` имеет имя `kl.BlobContainer`, но это имя можно изменить через переменную окружения `_BLOB_CONTAINER_BACKEND`. Эту переменную нужно задать как для процесса `BlobContainer`, так и для процессов, использующих динамические библиотеки.

Переменная окружения `_BLOB_CONTAINER_BACKEND` задает не только имя статически создаваемых IPC-каналов к процессу программы `BlobContainer`, но и имя службы, публикуемое на сервере имен, которое используется для динамического создания IPC-каналов к процессу программы `BlobContainer`. Это удобно использовать, когда запущено одновременно несколько процессов программы `BlobContainer` (например, с целью изоляции собственных динамических библиотек от сторонних), и разные процессы, использующие динамические библиотеки, должны взаимодействовать через IPC с разными процессами программы `BlobContainer`. В таком случае для разных процессов программы `BlobContainer` нужно задать разные значения переменной окружения `_BLOB_CONTAINER_BACKEND`, а затем использовать эти значения для переменной окружения `_BLOB_CONTAINER_BACKEND` процессов, использующих динамические библиотеки, выбирая конкретное значение в зависимости того, с каким именно процессом программы `BlobContainer` требуется динамически создать IPC-канал.

Пример использования переменной окружения `_BLOB_CONTAINER_BACKEND` в файле `init.yaml.in`:

```
entities:

- name: example.BlobContainer
  path: example_blob_container
  args:
  - "-v"
  env:
    _BLOB_CONTAINER_BACKEND: kl.custombc
@INIT_example_blob_container_ENTITY_CONNECTIONS@

- name: client.Client
  path: client
  env:
    _BLOB_CONTAINER_BACKEND: kl.custombc
@INIT_client_ENTITY_CONNECTIONS@

@INIT_EXTERNAL_ENTITIES@
```

Пример использования переменной окружения `_BLOB_CONTAINER_BACKEND` в `CMake`-командах:

```
set_target_properties (ExecMgrEntity PROPERTIES
EXTRA_ENV
"    _BLOB_CONTAINER_BACKEND: kl.custombc")

set_target_properties (dump_collector::entity PROPERTIES
EXTRA_ENV
"    _BLOB_CONTAINER_BACKEND: kl.custombc")
```

Если используется программа `BlobContainer`, то VFS, работающая с файлами динамических библиотек, должна быть отдельным процессом. Также должен быть создан IPC-канал от процесса программы `BlobContainer` к процессу VFS.

Сборка динамических библиотек

Для сборки динамических библиотек требуется использовать тулчейн, который поддерживает динамическую компоновку.

Чтобы выполнить сборку динамической библиотеки, нужно использовать следующую `CMake`-команду:

```
add_library(<имя цели сборки> SHARED [список путей к файлам исходного кода
библиотеки])
```

При использовании этой `CMake`-команды возникает ошибка, если тулчейн не поддерживает динамическую компоновку.

Также можно выполнить сборку динамической библиотеки следующей `CMake`-командой:

```
add_library(<имя цели сборки> [список путей к файлам исходного кода библиотеки])
```

При этом shell-команду `cmake` нужно вызвать с параметром `-D BUILD_SHARED_LIBS=YES`. (Если вызвать shell-команду `cmake` без параметра `-D BUILD_SHARED_LIBS=YES`, будет выполнена сборка статической библиотеки.)

Пример:

```
#!/bin/bash
...
cmake -G "Unix Makefiles" \
      -D CMAKE_BUILD_TYPE:STRING=Debug \
      -D CMAKE_TOOLCHAIN_FILE=$SDK_PREFIX/toolchain/share/toolchain-$TARGET.cmake \
      -D BUILD_SHARED_LIBS=YES \
      -B build \
      && cmake --build build --target kos-image
```

По умолчанию имя файла библиотеки совпадает с именем цели сборки, заданным через параметр `CMake`-команды `add_library()`. Имя файла библиотеки можно изменить, используя `CMake`-команду `set_target_properties()`. Это можно использовать, чтобы имя файла библиотеки было одинаковым для ее динамического и статического варианта.

Пример:

```
# Сборка статической библиотеки
add_library(somelib_static STATIC src/somesrc.cpp)
set_target_properties(somelib_static PROPERTIES OUTPUT_NAME "somelib")
# Переменная PLATFORM_SUPPORTS_DYNAMIC_LINKING имеет
# значение "истина", если используется динамическая
# компоновка. Если вызывается initialize_platform(FORCE_STATIC),
# эта переменная имеет значение "ложь".
if(PLATFORM_SUPPORTS_DYNAMIC_LINKING)
#   Сборка динамической библиотеки
    add_library(somelib_shared SHARED src/somesrc.cpp)
    set_target_properties(somelib_shared PROPERTIES OUTPUT_NAME "somelib")
endif()
```

Динамическая библиотека может быть скомпонована с другими статическими и динамическими библиотеками `CMake`-командой `target_link_libraries()`. При этом статические библиотеки должны быть собраны с флагом `-fPIC`. Этот флаг применяется при сборке статической библиотеки, если используется следующая `CMake`-команда:

```
set_property(TARGET <список имен целей сборки> PROPERTY POSITION_INDEPENDENT_CODE ON)
```

Добавление динамических библиотек в образ решения на базе KasperskyOS

Чтобы добавить динамические библиотеки в образ решения на базе KasperskyOS, нужно использовать параметры `PACK_DEPS_COPY_ONLY ON`, `PACK_DEPS_LIBS_PATH` и `PACK_DEPS_COPY_TARGET` в `CMake`-командах `build_kos_qemu_image()` и `build_kos_hw_image()`.

Пример:

```
set(RESOURCES ${CMAKE_SOURCE_DIR}/resources)
set(FSTAB ${RESOURCES}/fstab)
set(DISK_IMG ${CMAKE_CURRENT_BINARY_DIR}/ramdisk0.img)
set(RESOURCES_DIR ${CMAKE_CURRENT_SOURCE_DIR}/../resources)
set(EXT4_PART_DIR ${CMAKE_CURRENT_BINARY_DIR}/../system_hdd)

set_target_properties(${vfs_ENTITY} PROPERTIES
EXTRA_ARGS
"  - \"-f\"
  - \"fstab\""
EXTRA_ENV
"    ROOTFS: ramdisk0 / ext4 0"
${blkdev_ENTITY}_REPLACEMENT "${ramdisk_ENTITY};${sdcard_ENTITY}")

add_custom_target(copy-so)

add_custom_command(OUTPUT ${DISK_IMG}
       COMMAND ${CMAKE_COMMAND} -E copy_directory ${RESOURCES_DIR}/rootdir
${EXT4_PART_DIR}
       COMMAND mke2fs -v -d ${EXT4_PART_DIR} -t ext4 ${DISK_IMG} 40M
       DEPENDS copy-so
       COMMENT "Creating disk image '${DISK_IMG}' from files in '${EXT4_PART_DIR}'
...")

build_kos_hw_image(kos-image
                   ...
                   IMAGE_FILES ${ENTITIES_LIST} ${FSTAB} ${DISK_IMG}
                   PACK_DEPS_COPY_ONLY ON
                   PACK_DEPS_LIBS_PATH ${EXT4_PART_DIR}/lib
                   PACK_DEPS_COPY_TARGET copylibs)

if(PLATFORM_SUPPORTS_DYNAMIC_LINKING)
    add_dependencies(copy-so copylibs)
endif()
```

Динамические библиотеки, от которых зависят программы решения, будут добавлены в образ накопителя (например, с файловой системой ext4), который будет включен в образ решения.

> Динамические библиотеки, которые загружаются в память вызовом функции `dlopen()` интерфейса POSIX, не попадут в образ решения.

Система сборки выполняет следующие действия:

- Осуществляет поиск динамических библиотек и копирует эти библиотеки в директорию, путь к которой указан в параметре `PACK_DEPS_LIBS_PATH` `CMake`-команд `build_kos_qemu_image()` и `build_kos_hw_image()`. (Чтобы найденные динамические библиотеки попали в образ накопителя, эта директория должна находиться в файловой системе, которая будет помещена в образ накопителя.)

- Создает образ накопителя, который включает директорию с динамическими библиотеками.

  Чтобы создать образ накопителя, нужно использовать `CMake`-команду `add_custom_command()`. Цель, указанная в параметре `DEPENDS` `CMake`-команды `add_custom_command()`, означает создание образа накопителя. Цель, указанная в параметре `PACK_DEPS_COPY_TARGET` `CMake`-команд `build_kos_qemu_image()` и `build_kos_hw_image()`, означает копирование динамических библиотек. Чтобы образ накопителя был создан только после завершения копирования динамических библиотек, нужно использовать `CMake`-команду `add_dependencies()`.

- Добавляет образ накопителя в образ решения.

  Чтобы добавить образ накопителя в образ решения, нужно указать полный путь к образу накопителя в параметре `IMAGE_FILES` `CMake`-команд `build_kos_qemu_image()` и `build_kos_hw_image()`.

Формальные спецификации компонентов решения на базе KasperskyOS

При разработке решения создаются формальные спецификации его компонентов, которые формируют "картину мира" для модуля безопасности Kaspersky Security Module. *Формальная спецификация компонента решения на базе KasperskyOS* (далее *формальная спецификация компонента решения*) представляет собой систему IDL-, CDL-, EDL-описаний (IDL- и CDL-описания опциональны) этого компонента. Эти описания используются для автоматической генерации транспортного кода компонентов решения, а также исходного кода модуля безопасности и инициализирующей программы. Также формальные спецификации компонентов решения используются как исходные данные для описания политики безопасности решения.

> У ядра KasperskyOS так же, как и у компонентов решения, есть формальная спецификация (подробнее см. "Методы служб ядра KasperskyOS").

Каждый компонент решения соответствует EDL-описанию. С точки зрения формальной спецификации компонент решения – это контейнер компонентов, предоставляющих службы. Одновременно может использоваться несколько экземпляров одного компонента решения, то есть из одного исполняемого файла может быть запущено несколько процессов. Процессы, которые соответствуют одному и тому же EDL-описанию, являются процессами одного класса. EDL-описание задает имя класса процессов и параметры компонента верхнего уровня: предоставляемые службы с одним или несколькими интерфейсами, интерфейс безопасности и вложенные компоненты.

Каждый вложенный компонент соответствует CDL-описанию. Это описание задает имя компонента, предоставляемые службы, интерфейс безопасности, а также вложенные компоненты. Вложенные компоненты могут одновременно предоставлять службы, поддерживать интерфейс безопасности и являться контейнерами для других компонентов. Каждый вложенный компонент может предоставлять несколько служб с одним или несколькими интерфейсами.

Каждый интерфейс, включая интерфейс безопасности, определяется в IDL-описании. Это описание задает имя интерфейса, сигнатуры интерфейсных методов и типы данных для параметров интерфейсных методов. Данные, которые состоят из сигнатур интерфейсных методов и определений типов данных для параметров интерфейсных методов, называются пакетом.

Процессы, которые не предоставляют службы, могут быть только клиентами. Процессы, которые предоставляют службы, являются серверами, но и одновременно могут быть клиентами.

> Формальная спецификация компонента решения не определяет, как будет реализован этот компонент. То есть наличие компонентов в формальной спецификации компонента решения не означает, что эти компоненты будут присутствовать в архитектуре этого компонента решения.

Имена классов процессов, компонентов, пакетов и интерфейсов

Классы процессов, компоненты, пакеты и интерфейсы идентифицируются в IDL-, CDL-, EDL-описаниях по именам. В рамках решения на базе KasperskyOS имена классов процессов и имена компонентов образуют одно множество имен, а имена пакетов образуют другое множество имен. Эти два множества могут пересекаться. Множество имен пакетов включает в себя множество имен интерфейсов.

Имя класса процессов, компонента, пакета или интерфейса является ссылкой на IDL-, CDL- или EDL-файл, в котором это имя задано. Эта ссылка представляет собой путь к IDL-, CDL- или EDL-файлу (без расширения и точки перед ним) относительно директории, которая включена в набор директорий, где генераторы исходного кода выполняют поиск IDL-, CDL-, EDL-файлов. (Этот набор директорий задается параметрами `-I <путь к директории>`.) В качестве разделителя в описании пути используется точка.

Например, имя класса процессов `kl.core.NameServer` является ссылкой на EDL-файл `NameServer.edl`, который находится в KasperskyOS SDK по пути:

`sysroot-*-kos/include/kl/core`

При этом генераторы исходного кода должны быть настроены на поиск IDL-, CDL-, EDL-файлов в директории:

`sysroot-*-kos/include`

> Имя IDL-, CDL- или EDL-файла начинается с заглавной буквы и не может содержать символов подчеркивания `_`.

EDL-описание

EDL-описания помещаются в отдельные файлы `*.edl` и содержат декларации на языке EDL (Entity Definition Language):

1. **Имя класса процессов.** Используется декларация:

   ```
   entity <имя класса процессов>
   ```

2. [Опционально] **Список экземпляров компонентов.** Используется декларация:

   ```
   components {
       <имя экземпляра компонента : имя компонента>
       [...]
   }
   ```

   Каждый экземпляр компонента указывается отдельной строкой. Имя экземпляра компонента не может содержать символов подчеркивания `_`. Список может содержать несколько экземпляров одного компонента. Каждый экземпляр компонента в списке имеет уникальное имя.

3. [Опционально] **Интерфейс безопасности.** Используется декларация:

   ```
   security <имя интерфейса>
   ```

4. [Опционально] **Список служб.** Используется декларация:

   ```
   endpoints {
       <имя службы : имя интерфейса>
   ```

```
    [...]
}
```

Каждая служба указывается отдельной строкой. Имя службы не может содержать символов подчеркивания `_`. Список может содержать несколько служб с одинаковым интерфейсом. Каждая служба в списке имеет уникальное имя.

Язык EDL чувствителен к регистру символов.

В EDL-описании могут использоваться однострочные и многострочные комментарии.

> Интерфейс безопасности и предоставляемые службы могут задаваться в EDL-описании и в CDL-описании. Если при разработке компонента решения используются уже готовые составные части (например, в виде библиотек), которые сопровождаются CDL-описаниями, то целесообразно сослаться на них из EDL-описания через декларацию `components`. В противном случае можно описать все предоставляемые службы в EDL-описании. Кроме того, в EDL-описании и в каждом CDL-описании можно отдельно задать интерфейс безопасности.

Примеры EDL-файлов

Hello.edl

```
// Класс процессов, которые не содержат компонентов.
entity Hello
```

Signald.edl

```
/* Класс процессов, которые содержат
 * один экземпляр одного компонента. */
entity kl.Signald

components {
    signals : kl.Signals
}
```

LIGHTCRAFT.edl

```
/* Класс процессов, которые содержат
 * два экземпляра разных компонентов. */
entity kl.drivers.LIGHTCRAFT

components {
    KUSB : kl.drivers.KUSB
    KIDF : kl.drivers.KIDF
}
```

Downloader.edl

```
/* Класс процессов, которые не содержат
 * компонентов и предоставляют одну службу. */
entity updater.Downloader

endpoints {
```

```
    download : updater.Download
}
```

CDL-описание

CDL-описания помещаются в отдельные файлы `*.cdl` и содержат декларации на языке CDL (Component Definition Language):

1. **Имя компонента.** Используется декларация:

   ```
   component <имя компонента>
   ```

2. [Опционально] **Интерфейс безопасности.** Используется декларация:

   ```
   security <имя интерфейса>
   ```

3. [Опционально] **Список служб.** Используется декларация:

   ```
   endpoints {
       <имя службы : имя интерфейса>
       [...]
   }
   ```

   Каждая служба указывается отдельной строкой. Имя службы не может содержать символов подчеркивания `_`. Список может содержать несколько служб с одинаковым интерфейсом. Каждая служба в списке имеет уникальное имя.

4. [Опционально] **Список экземпляров вложенных компонентов.** Используется декларация:

   ```
   components {
       <имя экземпляра компонента : имя компонента>
       [...]
   }
   ```

   Каждый экземпляр компонента указывается отдельной строкой. Имя экземпляра компонента не может содержать символов подчеркивания `_`. Список может содержать несколько экземпляров одного компонента. Каждый экземпляр компонента в списке имеет уникальное имя.

Язык CDL чувствителен к регистру символов.

В CDL-описании могут использоваться однострочные и многострочные комментарии.

> В CDL-описании используется как минимум одна опциональная декларация. Если в CDL-описании не использовать ни одной опциональной декларации, то этому описанию будет соответствовать "пустой" компонент, который не предоставляет служб, не содержит вложенных компонентов и не поддерживает интерфейс безопасности.

Примеры CDL-файлов

KscProductEventsProvider.cdl

```
// Компонент предоставляет одну службу.
component kl.KscProductEventsProvider

endpoints {
    eventProvider : kl.IKscProductEventsProvider
}
```

KscConnectorComponent.cdl

```
// Компонент предоставляет несколько служб.
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

```
/* Компонент не предоставляет службы, поддерживает
 * интерфейс безопасности и содержит один экземпляр
 * другого компонента. */
component FsVerifier

security Approve

components {
    verifyComp : Verify
}
```

IDL-описание

IDL-описания помещаются в отдельные файлы `*.idl` и содержат декларации на языке IDL (Interface Definition Language):

1. **Имя пакета.** Используется декларация:

   ```
   package <имя пакета>
   ```

2. [Опционально] **Пакеты, из которых импортируются типы данных для параметров интерфейсных методов.** Используется декларация:

```
import <имя пакета>
```

3. [Опционально] **Определения типов данных для параметров интерфейсных методов**.

4. [Опционально] **Сигнатуры интерфейсных методов.** Используется декларация:

```
interface {
    <имя интерфейсного метода([параметры])>;
    [...]
}
```

   Каждая сигнатура метода указывается отдельной строкой. Имя метода не может содержать символов подчеркивания `_`. Каждый метод в списке имеет уникальное имя. Параметры методов разделяются на входные (`in`), выходные (`out`) и параметры для передачи сведений об ошибках (`error`). Порядок описания параметров важен: сначала указываются входные, затем выходные и, далее, error-параметры. Методы интерфейса безопасности не могут иметь выходные и error-параметры.

   Входные и выходные параметры передаются в IPC-запросах и IPC-ответах соответственно. Error-параметры передаются в IPC-ответах, если сервер не может корректно обработать соответствующие IPC-запросы.

   Сервер может информировать клиента об ошибках обработки IPC-запросов как через error-параметры, так и через выходные параметры интерфейсных методов. Если при возникновении ошибки сервер устанавливает флаг ошибки в IPC-ответе, то этот IPC-ответ содержит error-параметры и не содержит выходных параметров. В противном случае этот IPC-ответ содержит выходные параметры так же, как и при корректной обработке запросов. (Для установки флага ошибки в IPC-ответах используется макрос `nk_err_reset()`, определенный в заголовочном файле `nk/types.h` из состава KasperskyOS SDK.)

   Отправка IPC-ответа с установленным флагом ошибки и отправка IPC-ответа со снятым флагом ошибки являются разными видами событий для модуля безопасности Kaspersky Security Module. При описании политики безопасности решения это позволяет удобно разделять обработку событий, которые связаны с корректным и некорректным выполнением IPC-запросов. Если сервер не устанавливает флаг ошибки в IPC-ответах, то для обработки событий, связанных с некорректным выполнением IPC-запросов, модулю безопасности требуется проверять значения выходных параметров, которые сигнализируют об ошибках. (Клиент может проверить состояние флага ошибки в IPC-ответе, даже если соответствующий интерфейсный метод не содержит error-параметров. Для этого клиент использует макрос `nk_msg_check_err()`, определенный в заголовочном файле `nk/types.h` из состава KasperskyOS SDK.)

   Сигнатуры интерфейсных методов не могут импортироваться из других IDL-файлов.

Язык IDL чувствителен к регистру символов.

В IDL-описании могут использоваться однострочные и многострочные комментарии.

> В IDL-описании используется как минимум одна опциональная декларация. Если в IDL-описании не использовать ни одной опциональной декларации, то этому описанию будет соответствовать "пустой" пакет, который не задает ни интерфейсных методов, ни типов данных (в том числе из других IDL-описаний). Некоторые IDL-файлы из состава KasperskyOS SDK не описывают интерфейсные методы, а только содержат определения типов данных. Такие IDL-файлы используются только как экспортеры типов данных. Если пакет содержит описание интерфейсных методов, то имя интерфейса соответствует имени пакета.

Примеры IDL-файлов

```
Env.idl
```

```
package kl.Env

// Определения типов данных для параметров интерфейсного метода
typedef string<128> Name;
typedef string<256> Arg;
typedef sequence<Arg,256> Args;

// Интерфейс включает один метод.
interface {
    Read(in Name name, out Args args, out Args envs);
}
```

Kpm.idl

```
package kl.Kpm

// Импорт типов данных для параметров интерфейсных методов
import kl.core.Types

// Определение типа данных для параметров интерфейсных методов
typedef string<64> String;

/* Интерфейс включает несколько методов.
 * Часть методов не имеет параметров. */
interface {
    Shutdown();
    Reboot();
    PowerButtonPressedWait();
    TerminationSignalWait(in UInt32 entityId, in String entityName);
    EntityTerminated(in UInt32 entityId);
    Terminate(in UInt32 callingEntityId);
}
```

MessageBusSubs.idl

```
package kl.MessageBusSubs

// Импорт типов данных для параметров интерфейсного метода
import kl.MessageBusTypes

/* Интерфейс включает метод, который имеет
 * входной и выходные параметры, а также
 *  error-параметр.*/
interface {
    Wait(in ClientId id,
         out Message topic,
         out BundleId dataId,
         error ResultCode result);
}
```

WaylandTypes.idl

```
// Пакет содержит только определения типов данных.
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;
```

Типы данных в языке IDL

В языке IDL поддерживаются как примитивные, так и составные типы данных. Набор поддерживаемых составных типов включает объединения, структуры, массивы и последовательности.

Примитивные типы

В языке IDL поддерживаются следующие примитивные типы:

- `SInt8`, `SInt16`, `SInt32`, `SInt64` – знаковое целое число.
- `UInt8`, `UInt16`, `UInt32`, `UInt64` –беззнаковое целое число.
- `Handle` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора.
- `bytes<<размер в байтах>>` – байтовый буфер, представляющий собой область памяти с размером, не превышающим заданного числа байт.
- `string<<размер в байтах>>` – строковый буфер, представляющий собой байтовый буфер, последний байт которого является терминирующим нулем. Максимальный размер строкового буфера на единицу больше заданного из-за наличия дополнительного байта с терминирующим нулем.

Целочисленные литералы можно указывать в десятичном, шестнадцатеричном (например, `0x2f`, `0X2f`, `0x2F`, `0X2F`) или восьмеричном (например, `0O123`, `0o123`) представлении.

С помощью ключевого слова `const` можно определять именованные целочисленные константы, задавая их значения целочисленными литералами или целочисленными выражениями.

Примеры определений именованных целочисленных констант:

```
const UInt32 DeviceNameMax          = 0o100;
const UInt32 HandleTypeUserLast     = 0x0001FFFF;
const UInt32 MaxLogMessageSize = (2 << 3) ** 2;
const UInt32 MaxLogMessageCount = 100;
const UInt64 MaxLen = (MaxLogMessageSize + 4) * MaxLogMessageCount;
```

Именованные целочисленные константы можно использовать, чтобы избежать проблемы "магических чисел". К примеру, если в IDL-описании определены именованные целочисленные константы для кодов возврата интерфейсного метода, то при описании политики можно интерпретировать эти коды без дополнительных сведений. Также именованные целочисленные константы и целочисленные выражения могут применяться в определениях байтовых и строковых буферов, а также составных типов, чтобы задать размер данных или число элементов данных.

Конструкции `bytes<<размер в байтах>>` и `string<<размер в байтах>>` используются в определениях составных типов, сигнатурах интерфейсных методов и при создании псевдонимов типов, так как сами по себе они определяют анонимные типы (типы без имени).

Объединения

Объединение позволяет хранить данные разных типов в одной области памяти. В IPC-сообщении объединение снабжается дополнительным полем `tag`, позволяющим определить, какой именно член объединения используется.

Для определения объединения используется следующая конструкция:

```
union <имя типа> {
    <тип члена> <имя члена>;
    [...]
}
```

Пример определения объединения:

```
union ExitInfo {
    UInt32        code;
    ExceptionInfo exc;
}
```

Структуры

Для определения структуры используется следующая конструкция:

```
struct <имя типа> {
    <тип поля> <имя поля>;
    [...]
}
```

Пример определения структуры:

```
struct SessionEvqParams {
    UInt32 count;
    UInt32 align;
    UInt32 size;
}
```

Массивы

Для определения массива используется следующая конструкция:

```
array<<тип элементов, число элементов>>
```

Эта конструкция используется в определениях других составных типов, сигнатурах интерфейсных методов и при создании псевдонимов типов, так как сама по себе она определяет анонимный тип.

> Тип `Handle` можно использовать в качестве типа элементов массива, если этот массив не входит в другой составной тип данных. При этом суммарное число дескрипторов в IPC-сообщении не может превышать 255.

Последовательности

Последовательность представляет собой массив переменного размера. При определении последовательности указывается максимальное число ее элементов.

Для определения последовательности используется следующая конструкция:

```
sequence<<тип элементов, число элементов>>
```

Эта конструкция используется в определениях других составных типов, сигнатурах интерфейсных методов и при создании псевдонимов типов, так как сама по себе она определяет анонимный тип.

> Тип `Handle` нельзя использовать в качестве типа элементов последовательности.

Типы переменного и фиксированного размера

Типы `bytes`, `string` и `sequence` являются типами переменного размера, то есть при определении этих типов задается максимальное число элементов, а фактически может использоваться меньше, в том числе ноль. Данные типов `bytes`, `string` и `sequence` хранятся в арене IPC-сообщений. Остальные типы являются типами фиксированного размера. Данные типов фиксированного размера хранятся в фиксированной части IPC-сообщений.

Типы на основе составных типов

На основе составных типов могут быть определены другие составные типы. При этом определение массива или последовательности может быть вложено в определение другого типа.

Пример определения структуры с вложенными определениями массива и последовательности:

```
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;
    UInt64 e;
}
```

Определение объединения или структуры не может быть вложено в определение другого типа. Однако в определение типа могут быть включены уже определенные объединения и структуры. Это выполняется посредством указания в определении типа имен включаемых типов.

Пример определения структуры, включающей объединение и структуру:

```
union foo {
    UInt32 value1;
    UInt8 value2;
}

struct bar {
    UInt32 a;
    UInt8 b;
}

struct BazInfo {
    foo x;
    bar y;
}
```

Создание псевдонимов типов

Псевдонимы типов используются для повышения удобства работы с типами. Псевдонимы типов могут применяться, например, для того, чтобы задать типам с абстрактными именами мнемонические имена. Также назначение псевдонимов для анонимных типов позволяет получить именованные типы.

Для создания псевдонима типа используется следующая конструкция:

```
typedef <имя типа/определение анонимного типа> <псевдоним типа>
```

Пример создания мнемонических псевдонимов:

```
typedef UInt64 ApplicationId;
typedef Handle PortHandle;
```

Пример создания псевдонима определению массива:

```
typedef array<UInt8, 4> IP4;
```

Пример создания псевдонима определению последовательности:

```
const UInt32 MaxDevices = 8;
struct Device {
    string<32> DeviceName;
    UInt8 DeviceID;
}
typedef sequence<Device, MaxDevices> Devices;
```

Пример создания псевдонима определению объединения:

```
union foo {
    UInt32 value1;
```

```
    UInt8 value2;
}

typedef foo bar;
```

Определение анонимных типов в сигнатурах интерфейсных методов

Анонимные типы могут быть определены в сигнатурах интерфейсных методов.

Пример определения последовательности в сигнатуре интерфейсного метода:

```
const UInt8 DeviceCount = 8;

interface {
    Poll(in UInt32 timeout,
         out sequence<UInt32, DeviceCount / 2> report,
         out UInt32 count,
         out UInt32 rc);
}
```

Целочисленные выражения в языке IDL

Целочисленные выражения в языке IDL составляются из именованных целочисленных констант, целочисленных литералов, операторов (см. таблицу ниже) и группирующих круглых скобок.

Пример использования целочисленных выражений:

```
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);
}
```

Если при вычислении выражения возникнет целочисленное переполнение, генератор исходного кода, использующий IDL-файл, завершит работу с ошибкой.

Сведения об операторах целочисленных выражений в языке IDL

Синтаксис	Операция	Приоритет	Ассоциативность	Особенности
-a	Смена знака	1	Нет	Нет.
~a	Побитовое отрицание	1	Нет	Нет.
a ** b	Возведение в степень	2	Нет	Имеет следующие особенности: • Из-за отсутствия ассоциативности нужно использовать скобки при указании нескольких операторов

				подряд, чтобы задать порядок выполнения операций.
				Пример:
				`(a ** b) ** c`
				`a ** (b ** c)`
				• Показатель степени не должен быть отрицательным.

`a * b`	Умножение	3	Левая	Нет.
`a / b`	Целочисленное деление	3	Левая	Имеет следующие особенности: • Результатом операции является значение, полученное округлением вещественного результата деления до ближайшего меньшего целого. Например, *4 / 3 = 1*, *-4 / 3 = -2*. • Делитель не должен быть нулевым.
`a % b`	Остаток от деления	3	Левая	Имеет следующие особенности: • Знак результата операции соответствует знаку делителя. Например, *-5 % 2 = 1*, *5 % -2 = -1*. • Делитель не должен быть нулевым.
`a + b`	Сложение	4	Левая	Нет.
`a - b`	Вычитание	4	Левая	Нет.

`a << b`	Битовый сдвиг влево	2*	Нет	Имеет следующие особенности:
				• Приоритет ниже, чем у унарных операций, но несравним с приоритетами бинарных арифметических операций, поэтому в выражениях с бинарными арифметическими операциями нужно использовать скобки, чтобы задать порядок выполнения операций.
				Пример:
				`a << (b + c)`
				`(a << b) ** c`
				• Из-за отсутствия ассоциативности нужно использовать скобки при указании нескольких операторов подряд, чтобы задать порядок выполнения операций.
				Пример:

				`(a << b) << c`
				`a << (b << c)`
				• Значение сдвига должно быть в промежутке *[0; 63]*.

`a >> b`	Битовый сдвиг вправо	2*	Нет	Имеет следующие особенности:
				• Приоритет ниже, чем у унарных операций, но несравним с приоритетами бинарных арифметических операций, поэтому в выражениях с бинарными арифметическими операциями нужно использовать скобки, чтобы задать порядок выполнения операций.
				Пример:
				`a >> (b * c)`
				`(a >> b) / c`
				• Из-за отсутствия ассоциативности нужно использовать скобки при указании нескольких операторов подряд, чтобы задать порядок выполнения операций.
				Пример:
				`(a >> b) >> c`
				`a >> (b >> c)`
				• Значение сдвига должно быть в промежутке *[0; 63]*.

Описание политики безопасности решения на базе KasperskyOS

Описание политики безопасности решения на базе KasperskyOS (далее также *описание политики безопасности решения*, *описание политики*) представляет собой набор связанных между собой текстовых файлов с расширением `psl`, которые содержат декларации на языке PSL (Policy Specification Language). Одни файлы ссылаются на другие через декларацию включения, в результате чего образуется иерархия файлов с одним файлом верхнего уровня. Файл верхнего уровня специфичен для решения. Файлы нижнего и промежуточных уровней содержат части описания политики безопасности решения, которые могут быть специфичными для решения или могут быть повторно использованы в других решениях.

Часть файлов нижнего и промежуточных уровней поставляется в составе SDK KasperskyOS. Эти файлы содержат определения базовых типов данных и формальные описания моделей безопасности KasperskyOS. *Модели безопасности KasperskyOS* (далее *модели безопасности*) – это фреймворк для реализации политик безопасности решений на базе KasperskyOS. Файлы с формальными описаниями моделей безопасности ссылаются на файл с определениями базовых типов данных, которые используются в описаниях моделей.

Другая часть файлов нижнего и промежуточных уровней создается разработчиком описания политики, если какие-либо части описания политики требуется повторно использовать в других решениях. Также разработчик описания политики может помещать части описания политики в отдельные файлы для удобства редактирования.

Файл верхнего уровня ссылается на файлы с определениями базовых типов данных и описаниями моделей безопасности, которые применяются в той части политики безопасности решения, которая описана в этом файле. Также файл верхнего уровня ссылается на все файлы нижнего и промежуточных уровней, созданные разработчиком описания политики.

> Файл верхнего уровня, как правило, называется `security.psl`, но может иметь любое другое имя вида `*.psl`.

Общие сведения об описании политики безопасности решения на базе KasperskyOS

В упрощенном представлении описание политики безопасности решения на базе KasperskyOS состоит из привязок, которые ассоциируют описания событий безопасности с вызовами методов, предоставляемых объектами моделей безопасности. *Объект модели безопасности* – это экземпляр класса, определение которого является формальным описанием модели безопасности (в PSL-файле). Формальные описания моделей безопасности содержат сигнатуры *методов моделей безопасности*, которые определяют допустимость взаимодействий процессов между собой и с ядром KasperskyOS. Эти методы делятся на два вида:

- *Правила моделей безопасности* – это методы моделей безопасности, возвращающие результат "разрешено" или "запрещено". Правила моделей безопасности могут изменять контексты безопасности (о контексте безопасности см. "Управление доступом к ресурсам").

- *Выражения моделей безопасности* – это методы моделей безопасности, возвращающие значения, которые могут использоваться как входные данные для других методов моделей безопасности.

Объект модели безопасности предоставляет методы, специфичные для одной модели безопасности, и хранит параметры, используемые этими методами (например, начальное состояние конечного автомата или размер контейнера для каких-либо данных). Один объект может применяться для работы с несколькими ресурсами. (То есть не нужно создавать отдельный объект для каждого ресурса.) При этом контексты безопасности этих ресурсов будут независимы друг от друга. Также несколько объектов одной или разных моделей безопасности может применяться для работы с одним и тем же ресурсом. В этом случае разные объекты будут использовать контекст безопасности одного ресурса без взаимного влияния.

События безопасности – это сигналы об инициации взаимодействий процессов между собой и с ядром KasperskyOS. К событиям безопасности относятся следующие события:

- отправка IPC-запросов клиентами;

- отправка IPC-ответов серверами или ядром;

- инициация запусков процессов ядром или процессами;

- запуск ядра;

- обращения процессов к модулю безопасности Kaspersky Security Module через интерфейс безопасности.

События безопасности обрабатываются модулем безопасности.

Модели безопасности

В составе KasperskyOS SDK поставляются PSL-файлы, которые описывают следующие модели безопасности:

- Base – методы, реализующие простейшую логику;
- Pred – методы, реализующие операции сравнения;
- Bool – методы, реализующие логические операции;
- Math – методы, реализующие операции целочисленной арифметики;
- Struct – методы, обеспечивающие доступ к структурным элементам данных (например, доступ к параметрам интерфейсных методов, передаваемых в IPC-сообщениях);
- Regex – методы для валидации текстовых данных по регулярным выражениям;
- HashSet – методы для работы с одномерными таблицами, ассоциированными с ресурсами;
- StaticMap – методы для работы с двумерными таблицами типа "ключ–значение", ассоциированными с ресурсами;
- Flow – методы для работы с конечными автоматами, ассоциированными с ресурсами;
- Mic – методы для реализации *мандатного контроля целостности* (англ. Mandatory Integrity Control, MIC).

Обработка событий безопасности модулем безопасности Kaspersky Security Module

Модуль безопасности Kaspersky Security Module вызывает все методы (правила и выражения) моделей безопасности, связанные с произошедшим событием безопасности. Если все правила вернули результат "разрешено", модуль безопасности возвращает решение "разрешено". Если хотя бы одно правило вернуло результат "запрещено", модуль безопасности возвращает решение "запрещено".

Если хотя бы один метод, связанный с произошедшим событием безопасности, не может быть корректно выполнен, модуль безопасности возвращает решение "запрещено".

Если с произошедшим событием безопасности не связано ни одно правило, модуль безопасности возвращает решение "запрещено". То есть все взаимодействия компонентов решения между собой и с ядром KasperskyOS, которые явно не разрешены политикой безопасности решения, запрещены (принцип "Default Deny").

Синтаксис языка PSL

Базовые правила

1. Декларации могут располагаться в файле в любом порядке.
2. Одна декларация может быть записана в одну или несколько строк.

3. Язык PSL чувствителен к регистру символов.

4. Поддерживаются однострочные и многострочные комментарии:

```
/* Это комментарий
 * И это тоже */
// Ещё один комментарий
```

Типы деклараций

В языке PSL есть следующие типы деклараций:

- установка глобальных параметров политики безопасности решения;
- включение PSL-файлов в описание политики безопасности решения;
- включение EDL-файлов в описание политики безопасности решения;
- создание объектов моделей безопасности;
- привязка методов моделей безопасности к событиям безопасности;
- создание профилей аудита безопасности;
- создание тестов политики безопасности решения.

Установка глобальных параметров политики безопасности решения на базе KasperskyOS

Глобальными являются следующие параметры политики безопасности решения:

- *Execute-интерфейс*, через который ядро KasperskyOS обращается к модулю безопасности Kaspersky Security Module, чтобы сообщить о запуске ядра или об инициации запуска процесса ядром или другим процессом. Чтобы задать этот интерфейс, нужно использовать следующую декларацию:

  ```
  execute: kl.core.Execute
  ```

  В настоящее время в KasperskyOS поддерживается только один execute-интерфейс `Execute`, определенный в файле `kl/core/Execute.idl`. (Этот интерфейс состоит из одного метода `main`, который не имеет параметров и не выполняет никаких действий. Метод `main` зарезервирован для возможного использования в будущем.)

- [Опционально] Глобальный профиль аудита безопасности и начальный уровень аудита безопасности. (О профилях и уровне аудита безопасности см. "Создание профилей аудита безопасности".) Чтобы задать эти параметры, нужно использовать следующую декларацию:

  ```
  audit default = <имя профиля аудита безопасности> <уровень аудита безопасности>
  ```

Пример:

```
audit default = global 0
```

По умолчанию в качестве глобального используется пустой профиль аудита безопасности `empty`, описанный в файле `toolchain/include/nk/base.psl` из состава KasperskyOS SDK, и уровень аудита безопасности 0. Применение профиля аудита безопасности `empty` означает, что аудит безопасности не выполняется.

Включение PSL-файлов в описание политики безопасности решения на базе KasperskyOS

Чтобы включить в описание политики PSL-файл, нужно использовать следующую декларацию:

```
use <ссылка на PSL-файл._>
```

Ссылка на PSL-файл представляет собой путь к PSL-файлу (без расширения и точки перед ним) относительно директории, которая включена в набор директорий, где компилятор `nk-psl-gen-c` ищет PSL-, IDL-, CDL-, EDL-файлы. (Этот набор директорий задается параметрами `-I <путь к директории>` при запуске скрипта `makekss` или компилятора `nk-psl-gen-c`.) В качестве разделителя в описании пути используется точка. Декларация завершается последовательностью символов `._`.

Пример:

```
use policy_parts.flow_part._
```

Эта декларация включает файл `flow_part.psl`, который находится в директории `policy_parts`. Директория `policy_parts` должна находиться в одной из директорий, где компилятор `nk-psl-gen-c` выполняет поиск PSL-, IDL-, CDL-, EDL-файлов. Например, директория `policy_parts` может располагаться в одной директории с PSL-файлом, содержащим эту декларацию.

Включение PSL-файла с формальным описанием модели безопасности

Чтобы использовать методы требуемой модели безопасности, нужно включить в описание политики PSL-файл с формальным описанием этой модели. PSL-файлы с формальными описаниями моделей безопасности находятся в KasperskyOS SDK по пути:

`toolchain/include/nk`

Пример:

```
/* Включение файла base.psl с формальным описанием модели
 * безопасности Base */
use nk.base._

/* Включение файла flow.psl с формальным описанием модели
 *  безопасности Flow */
```

```
use nk.flow._
/* Компилятор nk-psl-gen-c должен быть настроен на поиск
 * PSL-, IDL-, CDL-, EDL-файлов в директории toolchain/include. */
```

Включение EDL-файлов в описание политики безопасности решения на базе KasperskyOS

Чтобы включить в описание политики EDL-файл для ядра KasperskyOS, нужно использовать следующую декларацию:

```
use EDL kl.core.Core
```

Чтобы включить в описание политики EDL-файл для программы (например, для драйвера или прикладной программы), нужно использовать следующую декларацию:

```
use EDL <ссылка на EDL-файл>
```

Ссылка на EDL-файл представляет собой путь к EDL-файлу (без расширения и точки перед ним) относительно директории, которая включена в набор директорий, где компилятор `nk-psl-gen-c` ищет PSL-, IDL-, CDL-, EDL-файлы. (Этот набор директорий задается параметрами `-I <путь к директории>` при запуске скрипта `makekss` или компилятора `nk-psl-gen-c`.) В качестве разделителя в описании пути используется точка.

Пример:

```
/* Включение файла UART.edl, который находится
 * в KasperskyOS SDK по пути sysroot-*-kos/include/kl/drivers. */
use EDL kl.drivers.UART
/* Компилятор nk-psl-gen-c должен быть настроен на поиск
 * PSL-, IDL-, CDL-, EDL-файлов в директории sysroot-*-kos/include. */
```

Компилятор `nk-psl-gen-c` находит IDL-, CDL-файлы через EDL-файлы, так как EDL-файлы содержат ссылки на соответствующие CDL-, IDL-файлы, а CDL-файлы содержат ссылки на соответствующие CDL-, IDL-файлы.

Создание объектов моделей безопасности

Чтобы вызывать методы требуемой модели безопасности, нужно создать объект этой модели безопасности.

Чтобы создать объект модели безопасности, нужно использовать следующую декларацию:

```
policy object <имя объекта модели безопасности : название модели безопасности> {
    [параметры объекта модели безопасности]
}
```

Имя объекта модели безопасности должно начинаться с маленькой буквы. Параметры объекта модели безопасности специфичны для модели безопасности. Описание параметров и примеры создания объектов разных моделей безопасности приведены в разделе "Модели безопасности KasperskyOS".

Привязка методов моделей безопасности к событиям безопасности

Чтобы создать привязку методов моделей безопасности к событию безопасности, нужно использовать следующую декларацию:

```
<вид события безопасности> [селекторы события безопасности] {
    [профиль аудита безопасности]
    <вызываемые методы моделей безопасности>
}
```

Вид события безопасности

Чтобы задать вид события безопасности, нужно использовать следующие спецификаторы:

- `request` – отправка IPC-запросов;
- `response` – отправка IPC-ответов;
- `error` – отправка IPC-ответов, содержащих сведения об ошибках;
- `security` – обращения процессов к модулю безопасности Kaspersky Security Module через интерфейс безопасности;
- `execute` – инициация запусков процессов или запуск ядра KasperskyOS.

> При взаимодействии процессов с модулем безопасности применяется механизм, отличный от IPC. Но при описании политики на обращения процессов к модулю безопасности можно смотреть как на передачу IPC-сообщений, так как процессы действительно передают модулю безопасности сообщения (в этих сообщениях не указывается приемник).
> Для запуска процессов не используется механизм IPC. Но когда инициируется запуск процесса, ядро обращается к модулю безопасности, сообщая сведения об инициаторе запуска и запускаемом процессе. Поэтому с точки зрения разработчика описания политики можно считать, что запуск процесса – это передача IPC-сообщения от инициатора запуска к запускаемому процессу. Также при запуске ядра можно считать, что ядро отправляет IPC-сообщение самому себе.

Селекторы события безопасности

Селекторы события безопасности позволяют уточнить описание события безопасности заданного вида. Можно использовать следующие селекторы:

- `src=<имя класса процессов/ядро>` – процессы заданного класса или ядро KasperskyOS являются источниками IPC-сообщений;
- `dst=<имя класса процессов/ядро>` – процессы заданного класса или ядро являются приемниками IPC-сообщений;

- `interface=<имя интерфейса>` – описывает следующие события безопасности:
  - клиенты пытаются использовать службы серверов или ядра с заданным интерфейсом;
  - процессы обращаются к модулю безопасности Kaspersky Security Module через заданный интерфейс безопасности;
  - серверы или ядро отправляют клиентам результаты использования служб с заданным интерфейсом;
- `component=<имя компонента>` – описывает следующие события безопасности:
  - клиенты пытаются использовать службы серверов или ядра, предоставляемые заданным компонентом;
  - серверы или ядро отправляют клиентам результаты использования служб, предоставляемых заданным компонентом;
- `endpoint=<квалифицированное имя службы>` – описывает следующие события безопасности:
  - клиенты пытаются использовать заданную службу серверов или ядра;
  - серверы или ядро отправляют клиентам результаты использования заданной службы;
- `method=<имя метода>` – описывает следующие события безопасности:
  - клиенты пытаются обратиться к серверам или ядру, вызывая заданный метод службы;
  - процессы обращаются к модулю безопасности, вызывая заданный метод интерфейса безопасности;
  - серверы или ядро отправляют клиентам результаты вызова заданного метода службы;
  - ядро сообщает о своем запуске модулю безопасности, вызывая заданный метод execute-интерфейса;
  - ядро инициирует запуски процессов, вызывая заданный метод execute-интерфейса;
  - процессы инициируют запуски других процессов, в результате чего ядро вызывает заданный метод execute-интерфейса.

Классы процессов, компоненты, экземпляры компонентов, интерфейсы, службы, методы должны называться так, как они называются в IDL-, CDL-, EDL-описаниях. Ядро должно называться `kl.core.Core`.

Квалифицированное имя службы является конструкцией вида `<путь к службе.имя службы>`. Путь к службе представляет собой последовательность разделенных точкой имен экземпляров компонентов, среди которых каждый последующий экземпляр компонента вложен в предыдущий, а последний предоставляет службу с заданным именем.

Для событий вида `security` нужно указывать квалифицированное имя метода интерфейса безопасности, если требуется использовать интерфейс безопасности, заданный в CDL-описании. (Если требуется использовать интерфейс безопасности, заданный в EDL-описании, указывать квалифицированное имя метода не нужно.) Квалифицированное имя метода интерфейса безопасности является конструкцией вида `<путь к интерфейсу безопасности.имя метода>`. Путь к интерфейсу безопасности представляет собой последовательность разделенных точкой имен экземпляров компонентов, среди которых каждый последующий экземпляр компонента вложен в предыдущий, а последний поддерживает интерфейс безопасности, который включает метод с заданным именем.

Если селекторы не указаны, участниками события безопасности могут быть любые процессы и ядро (кроме событий вида `security`, в которых ядро не может участвовать).

Можно использовать комбинации селекторов. При этом селекторы можно разделять запятыми.

На использование селекторов есть ограничения. Для событий безопасности вида `execute` нельзя использовать селекторы `interface`, `component` и `endpoint`. Для событий безопасности вида `security` нельзя использовать селекторы `dst`, `component`, `endpoint`.

Также есть ограничения на комбинации селекторов. Для событий безопасности видов `request`, `response` и `error` селектор `method` можно использовать только совместно с одним из селекторов `endpoint`, `interface`, `component` или их комбинацией. (Селекторы `method`, `endpoint`, `interface` и `component` должны быть согласованы, то есть метод, служба, интерфейс и компонент должны быть связаны между собой.) Для событий безопасности вида `request` селектор `endpoint` можно использовать только совместно с селектором `dst`. Для событий безопасности видов `response` и `error` селектор `endpoint` можно использовать только совместно с селектором `src`.

> Вид и селекторы события безопасности образуют описание события безопасности. События безопасности рекомендуется описывать максимально точно, чтобы разрешать только необходимые взаимодействия процессов между собой и с ядром. Если при обработке заданного события всегда проверяются IPC-сообщения одного и того же типа, то описание этого события является максимально точным.

Чтобы описанию события безопасности соответствовали IPC-сообщения одного типа, для этого описания должно выполняться одно из следующих условий:

- Для событий безопасности вида `request`, `response` и `error` однозначно определена цепочка "интерфейсный метод-служба-класс сервера или ядро". Например, описанию события безопасности `request dst=Server endpoint=net.Net method=Send` соответствуют IPC-сообщения одного типа, а описанию события безопасности `request dst=Server` соответствуют любые IPC-сообщения, отправляемые серверу `Server`.

- Для событий вида `security` указан метод интерфейса безопасности.

- Для событий вида `execute` указан метод execute-интерфейса.

  В настоящее время поддерживается только один фиктивный метод execute-интерфейса `main`. Этот метод используется по умолчанию, поэтому его можно не задавать через селектор `method`. Таким образом, любому описанию события безопасности вида `execute` соответствуют IPC-сообщения одного типа.

Профиль аудита безопасности

Чтобы задать профиль аудита безопасности, нужно использовать следующую конструкцию:

```
audit <имя профиля аудита безопасности>
```

Если профиль аудита безопасности не задан, используется глобальный профиль аудита безопасности.

Вызываемые методы моделей безопасности

Чтобы вызвать метод модели безопасности, нужно использовать следующую конструкцию:

```
[имя объекта модели безопасности.]<имя метода модели безопасности> <параметр>
```

В качестве параметра могут использоваться данные поддерживаемых в языке PSL типов. При этом нужно учитывать следующие особенности:

- Если у метода модели безопасности фактически нет параметра, то формально этот метод имеет параметр типа Unit, обозначаемый как `()`.
- Если параметром метода модели безопасности является словарь `{имя поля 1 : значение поля 1[, имя поля 2 : значение поля 2...]}`, то этот параметр не нужно заключать в круглые скобки.
- Если параметр метода модели безопасности не является словарем и не имеет тип Unit, то этот параметр нужно заключить в круглые скобки.

Можно вызвать один или несколько методов, используя один и тот же или разные объекты моделей безопасности. Правила моделей безопасности через параметр могут принимать значения, возвращаемые выражениями моделей безопасности.

> При обработке события безопасности модулем безопасности Kaspersky Security Module выражения вызываются перед правилами, поэтому выражения не получают изменений, сделанных правилами. Например, если в декларации привязки методов модели безопасности StaticMap к событиям безопасности сначала указано правило `set`, а затем для того же ресурса указано выражение `get_uncommited`, то выражение `get_uncommited` вернет значение ключа, которое было до обработки текущего события безопасности, а не то, которое задано правилом `set` при обработке текущего события безопасности. Значение ключа, заданное правилом `set` при обработке текущего события безопасности, может быть возвращено выражением `get_uncommited` только при обработки последующих событий безопасности, если в результате обработки текущего события безопасности модуль безопасности вернет решение "разрешено". Если в результате обработки текущего события безопасности модуль безопасности вернет решение "запрещено", то все изменения, сделанные правилами и выражениями, вызванными при обработке текущего события безопасности, будут отменены.

Метод модели безопасности (правило или выражение) через параметр может принимать параметры интерфейсных методов. (О получении доступа к параметрам интерфейсных методов см. "Модель безопасности Struct"). Также метод модели безопасности через параметр может принимать значения SID процессов и ядра KasperskyOS, которые задаются ключевыми словами `src_sid` и `dst_sid`. Первое означает SID процесса (или ядра), который является источником IPC-сообщения. Второе означает SID процесса (или ядра), который является приемником IPC-сообщения (при обращениях к модулю безопасности Kaspersky Security Module `dst_sid` использовать нельзя).

> Для вызова некоторых методов моделей безопасности можно не указывать имя объекта модели безопасности. Также часть методов моделей безопасности нужно вызывать, используя операторы, а не конструкцию вызова. Подробнее о методах моделей безопасности см. "Модели безопасности KasperskyOS".

Вложенные конструкции для привязки методов моделей безопасности к событиям безопасности

В одной декларации можно создать привязку методов моделей безопасности к разным событиям безопасности одного вида. Для этого нужно использовать match-секции, которые представляют собой конструкции следующего вида:

```
match <селекторы события безопасности> {
    [профиль аудита безопасности]
```

```
        <вызываемые методы моделей безопасности>
    }
```

Match-секции могут быть вложены в другую match-секцию. Match-секция использует одновременно свои селекторы события безопасности и селекторы события безопасности уровня декларации и всех match-секций, которые "оборачивают" эту match-секцию. Также match-секция применяет по умолчанию профиль аудита безопасности своего контейнера (match-секции предыдущего уровня или уровня декларации), но можно задать отдельный профиль аудита безопасности для match-секции.

Также в одной декларации можно задать различные варианты обработки события безопасности в зависимости от условий, при которых это событие наступило (например, от состояния конечного автомата, ассоциированного с ресурсом). Для этого нужно использовать условные секции, которые являются элементами следующей конструкции:

```
    choice <вызов выражения модели безопасности, проверяющего выполнение условий> {
        "<условие 1>" : [{] // Условная секция 1
            <вызываемые методы моделей безопасности>
        [}]
        "<условие 2>" : ... // Условная секция 2
    ...
        _             : ... // Условная секция, если ни одно условие не выполняется.
    }
```

Конструкцию `choice` можно использовать внутри match-секции. Условная секция использует селекторы события безопасности и профиль аудита безопасности своего контейнера.

Если при обработке события безопасности выполняется сразу несколько условий, описанных в конструкции `choice`, то срабатывает только одна условная секция, соответствующая первому в списке истинному условию.

> В качестве выражения, проверяющего выполнение условий в конструкции `choice`, можно использовать только те выражения, которые предназначены специально для этого. Некоторые модели безопасности содержат такие выражения (подробнее см. "Модели безопасности KasperskyOS").
> В качестве условий можно использовать только текстовые и целочисленные литералы, значения логического типа и символ `_`, обозначающий всегда истинное условие.

Примеры привязок методов моделей безопасности к событиям безопасности

См. "Примеры привязок методов моделей безопасности к событиям безопасности", "Примеры описаний простейших политик безопасности решений на базе KasperskyOS", "Модели безопасности KasperskyOS".

Создание профилей аудита безопасности

Аудит безопасности (далее также *аудит*) представляет собой следующую последовательность действий. Модуль безопасности Kaspersky Security Module сообщает ядру KasperskyOS сведения о решениях, принятых этим модулем. Затем ядро передает эти данные системной программе `Klog`, которая декодирует их и передает системной программе `KlogStorage` (передача данных осуществляется через IPC). Последняя направляет полученные данные в стандартный вывод (или стандартный вывод ошибок) либо записывает в файл.

Данные аудита безопасности (далее *данные аудита*) – это сведения о решениях модуля безопасности Kaspersky Security Module, которые включают сами решения ("разрешено" или "запрещено"), описания событий безопасности, результаты вызовов методов моделей безопасности, а также данные о некорректности IPC-сообщений. Данные о вызовах выражений моделей безопасности входят в данные аудита так же, как и данные о вызовах правил моделей безопасности.

Для выполнения аудита безопасности нужно ассоциировать объекты моделей безопасности с профилем (профилями) аудита безопасности. *Профиль аудита безопасности* (далее также *профиль аудита*) объединяет в себе *конфигурации аудита безопасности* (далее также *конфигурации аудита*), каждая из которых задает объекты моделей безопасности, покрываемые аудитом, а также условия выполнения аудита. Можно задать глобальный профиль аудита (подробнее см. "Установка глобальных параметров политики безопасности решения на базе KasperskyOS") и/или назначить профиль (профили) аудита на уровне привязок методов моделей безопасности к событиям безопасности, и/или назначить профиль (профили) аудита на уровне match-секций (подробнее см. "Привязка методов моделей безопасности к событиям безопасности").

> Независимо от того, используются профили аудита или нет, данные аудита содержат сведения о решениях "запрещено", которые приняты модулем безопасности Kaspersky Security Module при некорректности IPC-сообщений и обработке событий безопасности, не связанных ни с одним правилом моделей безопасности.

Чтобы создать профиль аудита безопасности, нужно использовать следующую декларацию:

```
audit profile <имя профиля аудита безопасности> =
  { <уровень аудита безопасности> :
      // Конфигурация аудита безопасности
      { <имя объекта модели безопасности> :
          { kss : <условия выполнения аудита безопасности, связанные с результатами
                   вызовов методов модели безопасности>
         [, условия выполнения аудита безопасности, специфичные для модели
безопасности]
          }
    [ ,...]
      }
[ ,...]
  }
```

Уровень аудита безопасности

Уровень аудита безопасности (далее *уровень аудита*) является глобальным параметром политики безопасности решения и представляет собой беззнаковое целое число, которое задает активную конфигурацию аудита безопасности. (Слово "уровень" здесь означает вариант конфигурации и не предполагает обязательной иерархии.) Уровень аудита можно изменять в процессе работы модуля безопасности Kaspersky Security Module. Для этого нужно использовать специальный метод модели безопасности `Base`, вызываемый при обращении процессов к модулю безопасности через интерфейс безопасности (подробнее см. "Модель безопасности Base"). Начальный уровень аудита задается совместно с глобальным профилем аудита (подробнее см. "Установка глобальных параметров политики безопасности решения на базе KasperskyOS"). В качестве глобального можно явно назначить пустой профиль аудита `empty`.

В профиле аудита можно задать несколько конфигураций аудита. В разных конфигурациях можно покрыть аудитом разные объекты моделей безопасности и применить разные условия выполнения аудита. Конфигурации аудита в профиле соответствуют разным уровням аудита. Если в профиле нет конфигурации аудита, соответствующей текущему уровню аудита, модуль безопасности задействует конфигурацию, которая соответствует ближайшему меньшему уровню аудита. Если в профиле нет конфигурации аудита для уровня аудита, равного или ниже текущего, модуль безопасности не будет использовать этот профиль (то есть аудит по этому профилю не будет выполняться).

Возможность изменять уровень аудита можно использовать, например, чтобы регулировать детализацию аудита. Чем выше уровень аудита, тем выше детализация. То есть при более высоком уровне аудита задействуются конфигурации аудита, при которых больше объектов моделей безопасности покрывается аудитом и/или меньше ограничений применяется в условиях выполнения аудита. Также, изменяя уровень аудита, можно переключать аудит с одного множества логически связанных объектов моделей безопасности на другое. То есть, например, при низком уровне аудита задействуются конфигурации аудита, при которых покрываются аудитом объекты моделей безопасности, связанные с драйверами; при среднем уровне аудита задействуются конфигурации аудита, при которых покрываются аудитом объекты моделей безопасности, связанные с сетевой подсистемой; при высоком уровне аудита задействуются конфигурации аудита, при которых покрываются аудитом объекты моделей безопасности, связанные с прикладными программами.

Имя объекта модели безопасности

Имя объекта модели безопасности указывается, чтобы методы, которые предоставляются этим объектом, могли быть покрыты аудитом. Эти методы будут покрыты аудитом при их вызовах, если условия выполнения аудита будут соблюдены.

> Сведения о решениях модуля безопасности Kaspersky Security Module, содержащиеся в данных аудита, включают как общее решение модуля безопасности, так и результаты вызовов отдельных методов моделей безопасности, покрытых аудитом. Чтобы сведения о решении модуля безопасности попали в данные аудита, нужно, чтобы по крайней мере один метод, вызванный при обработке события безопасности, был покрыт аудитом.
> Имена объектов моделей безопасности, как и имена методов, предоставляемых этими объектами, попадают в данные аудита.

Условия выполнения аудита безопасности

Условия выполнения аудита безопасности нужно задать отдельно для каждого объекта модели безопасности.

Чтобы задать условия выполнения аудита, связанные с результатами вызовов методов моделей безопасности, нужно использовать следующие конструкции:

- `["granted"]` – аудит выполняется, если правила возвращают результат "разрешено", выражения выполняются корректно.
- `["denied"]` – аудит выполняется, если правила возвращают результат "запрещено", выражения выполняются некорректно.
- `["granted", "denied"]` – аудит выполняется независимо от того, какой результат возвращают правила, и корректно ли выполняются правила.
- `[]` – аудит не выполняется.

Условия выполнения аудита, специфичные для моделей безопасности, задаются конструкциями, специфичными для этих моделей (подробнее см. "Модели безопасности KasperskyOS"). Эти условия применяются как к правилам, так и к выражениям. Например, таким условием может быть состояние конечного автомата.

Профиль аудита безопасности для тракта аудита безопасности

Тракт аудита безопасности включает ядро, а также процессы `Klog` и `KlogStorage`, которые соединены IPC-каналами по схеме "ядро – `Klog` – `KlogStorage`". Методы моделей безопасности, которые связаны с передачей данных аудита через этот тракт, не должны покрываться аудитом. В противном случае это приведет к лавинообразному росту данных аудита, так как передача данных будет порождать новые данные.

Чтобы "подавить" аудит, заданный профилем более широкой области действия (например, глобальным или профилем на уровне привязки методов моделей безопасности к событиям безопасности), нужно назначить пустой профиль аудита `empty` на уровне привязки методов моделей безопасности к событиям безопасности или на уровне match-секции.

Примеры профилей аудита безопасности

См. "Примеры профилей аудита безопасности".

Создание и выполнение тестов политики безопасности решения на базе KasperskyOS

Тестирование политики безопасности решения выполняется, чтобы проверить, разрешает ли политика то, что должна разрешать, и запрещает ли она то, что должна запрещать.

Чтобы создать набор тестов политики безопасности решения, нужно использовать декларацию:

```
assert ["название набора тестов"] {
// Конструкции на языке PAL (Policy Assertion Language)
    [setup {<начальная часть тестов>}]
    sequence ["название теста"] {<основная часть теста>}
    [...]
    [finally {<конечная часть тестов>}]
}
```

Можно создать несколько наборов тестов, используя несколько таких деклараций.

Набор тестов опционально включает начальную часть тестов и/или конечную часть тестов. Выполнение каждого теста из набора начинается с того, что описано в начальной части, и завершается тем, что описано в конечной части. Это позволяет не описывать повторяющиеся начальные и/или конечные части тестов в каждом тесте.

> После выполнения каждого теста все изменения в модуле безопасности Kaspersky Security Module, связанные с выполнением этого теста, "откатываются".

Каждый тест включает один или несколько тестовых примеров.

Тестовые примеры

Тестовый пример ассоциирует описание события безопасности и значения параметров интерфейсного метода с ожидаемым решением модуля безопасности Kaspersky Security Module. Если фактическое решение модуля безопасности совпадает с ожидаемым, тестовый пример проходит, иначе не проходит.

Когда выполняется тест, тестовые примеры выполняются в той последовательности, в которой они описаны. То есть осуществляется проверка, как модуль безопасности обрабатывает последовательность событий безопасности.

Если все тестовые примеры в тесте проходят, тест проходит. Если хотя бы один тестовый пример в тесте не проходит, тест не проходит. Выполнение теста завершается на первом тестовом примере, который не проходит. Каждый тест из набора выполняется независимо от того, прошел или не прошел предыдущий тест.

Описание тестового примера на языке PAL представляет собой следующую конструкцию:

```
[<ожидаемое решение модуля безопасности> ["название тестового примера"]]
<вид события безопасности> <селекторы события безопасности>
[{значения параметров интерфейсного метода}]
```

В качестве ожидаемого решения модуля безопасности можно указать значение `grant` ("разрешено"), `deny` ("запрещено") или `any` ("любое решение"). Можно не указывать ожидаемое решение модуля безопасности. По умолчанию ожидается решение "разрешено". Если указано значение `any`, решение модуля безопасности не влияет на то, проходит тестовый пример или нет. В этом случае тестовый пример может не пройти из-за ошибок обработки IPC-сообщения модулем безопасности (например, при некорректной структуре IPC-сообщения).

Название тестового примера можно указать, если только указано ожидаемое решения модуля безопасности.

О видах и селекторах событий безопасности, а также об ограничениях использования селекторов см. "Привязка методов моделей безопасности к событиям безопасности". Селекторы должны обеспечивать, чтобы описанию события безопасности соответствовали IPC-сообщения одного типа. (В привязках методов моделей безопасности к событиям безопасности селекторы могут этого не обеспечивать.)

В описаниях событий безопасности вместо имени класса процессов (и ядра KasperskyOS) нужно указывать SID. Исключение составляют события вида `execute`, при наступлении которых SID запускаемого процесса (или ядра) неизвестен. Чтобы сохранить SID процесса или ядра в переменную, нужно использовать оператор `<-` в описании тестового примера вида:

```
<имя переменной> <- execute dst=<имя класса процессов/ядро> ...
```

Переменной будет присвоено значение SID, даже если запуск процесса заданного класса (или ядра) запрещен тестируемой политикой, но решение "запрещено" является ожидаемым.

В языке PAL поддерживаются сокращенные формы описаний событий безопасности:

- `security: <SID процесса> ! <квалифицированное имя метода интерфейса безопасности>` соответствует `security src=<SID процесса> method= <квалифицированное имя метода интерфейса безопасности>`.
- `request: <SID клиента> ~> <SID сервера/ядра> : <квалифицированное имя службы.имя метода>` соответствует `request src=<SID клиента> dst=<SID сервера/ядра> endpoint=<квалифицированное имя службы> method=<имя метода>`.

- `response: <SID клиента> <~ <SID сервера/ядра> : <квалифицированное имя службы.имя метода>` соответствует `response src=<SID сервера/ядра> dst=<SID клиента> endpoint=<квалифицированное имя службы> method=<имя метода>`.

Значения параметров интерфейсного метода должны быть заданы для всех видов событий безопасности, кроме `execute`. Если у интерфейсного метода нет параметров, нужно указать `{}`. Для событий безопасности вида `execute` нельзя указывать `{}`.

Параметры интерфейсного метода и их значения нужно задавать разделенными запятой конструкциями следующего вида:

```
<имя параметра> : <значение>
```

Имена и типы параметров должны соответствовать IDL-описанию. Порядок следования параметров не важен.

Пример задания значений параметров:

```
{ param1 : 23, param2 : "bar", param3 : { collection : [5,7,12], filehandle : 15 },
param4 : { name : ["foo", "baz" } }
```

В этом примере через параметр `param1` передается число. Через параметр `param2` передается строковый буфер. Через параметр `param3` передается структура, состоящая из двух полей. Поле `collection` содержит массив или последовательность из трех числовых элементов. Поле `filehandle` содержит SID. Через параметр `param4` передается объединение или структура с одни полем. Поле `name` содержит массив или последовательность из двух строковых буферов.

> В настоящее время в качестве значения параметра типа `Handle` можно указывать только SID, а возможности указать SID совместно с маской прав дескриптора нет. Поэтому нельзя тестировать политику безопасности решения в тех случаях, когда маски прав дескрипторов влияют на решения модуля безопасности.

Значения параметров (элементов параметров) можно не указывать. В этом случае автоматически применяются значения по умолчанию, соответствующие IDL-типам параметров (элементов параметров):

- для числовых типов и типа `Handle` – ноль;
- для байтовых или строковых буферов – байтовый или строковый буфер нулевого размера;
- для последовательностей – последовательность с нулевым числом элементов;
- для массивов – массив элементов со значениями по умолчанию;
- для структур – структура, состоящая из полей со значениями по умолчанию;
- для объединений – значение по умолчанию, соответствующее первому члену объединения.

Пример применения значения по умолчанию для параметра и элемента параметра:

```
/* Параметр указан. */
request src=x dst=y endpoint=e method=m { name : { firstname: "a", lastname: "b" } }
```

```
/* Параметр не указан. В качестве значения параметра name будет применена структура,
 * состоящая из двух строковых буферов нулевого размера.*/
request src=x dst=y endpoint=e method=m {}
/* Элемент параметра не указан. В качестве значения элемента параметра lastname будет
 * применен строковый буфер нулевого размера.*/
request src=x dst=y endpoint=e method=m { name : { firstname: "a"} }
```

Примеры тестов

См. "Примеры тестов политик безопасности решений на базе KasperskyOS".

Тестовая процедура

Тестовая процедура включает следующие шаги:

1. Сохранить тесты в одном или нескольких PSL-файлах (`*.psl` или `*.psl.in`).

2. Добавить `CMake`-команду `add_kss_pal_qemu_tests()` в один из файлов `CMakeLists.txt` проекта.

   Через параметр `PSL_FILES` нужно задать пути к PSL-файлам с тестами. Через параметр `DEPENDS` нужно задать `CMake`-цели, в результате выполнения которых IDL-, CDL-, EDL-файлы, от которых зависят PSL-файлы, будут помещены в те директории, где компилятор `nk-psl-gen-c` сможет их найти. Если используются файлы `*.psl.in`, через параметр `ENTITIES` нужно задать имена классов процессов системных программ. (Эти системные программы входят в состав решения на базе KasperskyOS, для которого нужно выполнить тестирование политики безопасности.)

   Пример использования `CMake`-команды `add_kss_pal_qemu_tests()` в файле `einit/CMakeLists.txt`:

   ```
   add_kss_pal_qemu_tests (
     PSL_FILES src/security.psl.in
     DEPENDS kos-qemu-image
     ENTITIES ${ENTITIES})
   ```

3. Собрать и выполнить тесты.

   Нужно запустить Bash-скрипт сборки `cross-build.sh` с параметром `--target pal-test<N>`, где *N* – индекс PSL-файла в списке PSL-файлов, заданных через параметр `PSL_FILES` `CMake`-команды `add_kss_pal_qemu_tests()` на шаге 2. Например, если указать `--target pal-test0`, будет создан и запущен на QEMU образ решения на базе KasperskyOS, который соответствует первому PSL-файлу, заданному через параметр `PSL_FILES` `CMake`-команды `add_kss_pal_qemu_tests()`. (Вместо прикладных и системных программ это решение будет содержать программу, выполняющую тесты.)

   Пример:

   ```
   ./cross-build.sh --target pal-test0
   ```

Пример результатов тестирования:

```
[==========] Running 4 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 4 tests from KSS
[ RUN      ] KSS.KssUnitTest_flow_normal
[       OK ] KSS.KssUnitTest_flow_normal (6 ms)
```

```
[ RUN      ] KSS.KssUnitTest_flow_ping_must_be_first
/home/work/build/stat/build/install/examples/ping/build/einit/
pal-test/gen_security.psl.test.c:9742: Failure
Expected equality of these values:
  rc
    Which is: -1
  NK_EOK
    Which is: 0
gen_security.psl:116: expect grant

[  FAILED  ] KSS.KssUnitTest_flow_ping_must_be_first (8 ms)
[ RUN      ] KSS.KssUnitTest_flow_ping_ping_is_deny
[       OK ] KSS.KssUnitTest_flow_ping_ping_is_deny (4 ms)
[ RUN      ] KSS.KssUnitTest_flow_test_deny
[       OK ] KSS.KssUnitTest_flow_test_deny (1 ms)
[----------] 4 tests from KSS (29 ms total)

[----------] Global test environment tear-down
[==========] 4 tests from 1 test suite ran. (42 ms total)
[  PASSED  ] 3 tests.
[  FAILED  ] KSS.KssUnitTest_flow_ping_must_be_first (8 ms)
```

Результаты тестирования содержат сведения о том, прошел или не прошел каждый тест. Если тест не прошел, то указываются сведения о размещении описания непрошедшего тестового примера в PSL-файле.

Типы данных в языке PSL

Типы данных, поддерживаемые в языке PSL, приведены в таблице ниже.

Типы данных в языке PSL

Обозначения типов	Описание типов
`UInt8`, `UInt16`, `UInt32`, `UInt64`	Беззнаковое целое число
`SInt8`, `SInt16`, `SInt32`, `SInt64`	Знаковое целое число
`Boolean`	Логический тип Логический тип включает два значения: `true` и `false`.
`Text`	Текстовый тип
`()`	Тип `Unit` Тип `Unit` включает одно неизменяемое значение. Используется как заглушка в случаях, когда синтаксис языка PSL требует указать какие-либо данные, но фактически эти данные не требуются. Например, тип Unit можно использовать, чтобы объявить метод, который не имеет параметров (аналогично тому, как тип `void` используется в C/C++).

`"[тип]"`	Текстовый литерал
	Текстовый литерал включает одно неизменяемое текстовое значение.
	Примеры определений текстовых литералов:
	`""`
	`"granted"`

`<тип>`	Целочисленный литерал
	Целочисленный литерал включает одно неизменяемое целочисленное значение.
	Примеры определений числовых литералов:
	`12`
	`-5`
	`0xFFFF`

`<тип 1 | тип 2> [| ...]`	Вариантный тип
	Вариантный тип объединяет два и более типов и может выступать в роли любого из них.
	Примеры определений вариантных типов:
	`Boolean | ()`
	`UInt8 | UInt16 | UInt32 | UInt64`
	`"granted" | "denied"`

`{ [имя поля : тип поля]`
`[, ...]`
`}`
	Словарь
	Словарь состоит из полей одного или нескольких типов. Словарь может быть пустым.
	Примеры определений словарей:
	`{}`
	`{ handle : Handle`
	`, rights : UInt32`
	`}`

`[[тип] [, ...]]`	Кортеж
	Кортеж состоит из полей одного или нескольких типов, расположенных в порядке перечисления типов. Кортеж может быть пустым.
	Примеры определений кортежей:
	`[]`
	`["granted"]`
	`[Boolean, Boolean]`

`Set<<тип элементов>>`	Множество
	Множество включает ноль и более уникальных элементов одного типа.
	Примеры определений множеств:
	`Set<"granted" | "denied">`
	`Set<Text>`

`List<<тип элементов>>`	Список
	Список включает ноль и более элементов одного типа.
	Примеры определений списков:
	`List<Boolean>`
	`List<Text | ()>`

`Map<<тип ключа, тип значения>>`	Ассоциативный массив Ассоциативный массив включает ноль и более записей типа "ключ-значение" с уникальными ключами. Пример определения ассоциативного массива:

	`Map<UInt32, UInt32>`
`Array<<тип элементов, число элементов>>`	Массив Массив включает заданное число элементов одного типа. Пример определения массива: `Array<UInt8, 42>`
`Sequence<<тип элементов, число элементов>>`	Последовательность Последовательность включает от ноля до заданного числа элементов одного типа. Пример определения последовательности: `Sequence<SInt64, 58>`

Псевдонимы некоторых типов PSL

В файле `nk/base.psl` из состава KasperskyOS SDK определены типы данных, которые используются как типы параметров (или структурных элементов параметров) и возвращаемых значений для методов разных моделей безопасности. Псевдонимы и определения этих типов приведены в таблице ниже.

Псевдонимы и определения некоторых типов данных в языке PSL

Псевдоним типа	Определение типа
`Unsigned`	Беззнаковое целое число `UInt8 | UInt16 | UInt32 | UInt64`
`Signed`	Знаковое целое число `SInt8 | SInt16 | SInt32 | SInt64`
`Number`	Целое число `Unsigned | Signed`
`ScalarLiteral`	Скалярный литерал `() | Boolean | Number`
`Literal`	Литерал `ScalarLiteral | Text`
`Sid`	Тип идентификатора безопасности SID `UInt32`
`Handle`	Тип идентификатора безопасности SID `Sid`

`HandleDesc`	Словарь, содержащий поля для SID и маски прав дескриптора
	`{ handle : Handle`
	`, rights : UInt32`
	`}`

`Cases`	Тип данных, принимаемых выражениями моделей безопасности, вызываемыми в конструкции `choice` для проверки выполнения условий `List<Text | ()>`
`KSSAudit`	Тип данных, задающих условия выполнения аудита безопасности `Set<"granted" | "denied">`

Отображение типов IDL на типы PSL

Для описания параметров интерфейсных методов используются типы данных языка IDL. Входные данные для методов моделей безопасности имеют типы из языка PSL. Набор типов данных в языке IDL отличается от набора типов данных в языке PSL. Поскольку параметры интерфейсных методов, передаваемые в IPC-сообщениях, могут использоваться как входные данные для методов моделей безопасности, разработчику описания политики нужно понимать, как типы IDL отображаются на типы PSL.

Целочисленные типы IDL отображаются на целочисленные типы PSL, а также на вариантные типы PSL, объединяющие эти целочисленные типы (в том числе с другими типами). Например, знаковые целочисленные типы IDL отображаются на тип `Signed` в PSL, целочисленные типы IDL отображаются на тип `ScalarLiteral` в PSL.

Тип `Handle` в IDL отображается на тип `HandleDesc` в PSL.

Объединения и структуры IDL отображаются на словари PSL.

Массивы и последовательности IDL отображаются на массивы и последовательности PSL соответственно.

Строковые буферы в IDL отображаются на текстовый тип PSL.

В настоящее время байтовые буферы в IDL не отображаются на типы PSL. Соответственно, данные, содержащиеся в байтовых буферах, не могут использоваться как входы для методов моделей безопасности.

Примеры привязок методов моделей безопасности к событиям безопасности

> Перед тем как рассматривать примеры, нужно ознакомиться со сведениями о модели безопасности Base.

Обработка инициации запусков процессов

```
/* Ядру KasperskyOS и любому процессу
 * в решении разрешено запускать любой
 * процесс. */
execute { grant () }

/* Ядру разрешено запускать процесс
 * класса Einit. */
execute src=kl.core.Core, dst=Einit { grant () }

/* Процессу класса Einit разрешено
 * запускать любой процесс в решении. */
execute src=Einit { grant () }
```

Обработка запуска ядра KasperskyOS

```
/* Ядру KasperskyOS разрешено запускаться.
 * (Эта привязка нужна, чтобы сообщить модулю
 * безопасности SID ядра. Ядро запускается независимо
 * от того, разрешено ли это политикой безопасности решения
```

```
 * или нет. Если политика безопасности решения запрещает
 * запуск ядра, после запуска ядро прекратит свое
 * исполнение.) */
execute src=kl.core.Core, dst=kl.core.Core { grant () }
```

Обработка отправки IPC-запросов

```
/* Любому клиенту в решении разрешено обращаться к
 * любому серверу и ядру KasperskyOS. */
request { grant () }

/* Клиенту класса Client разрешено обращаться
 * к любому серверу в решении и ядру. */
request src=Client { grant () }

/* Любому клиенту в решении разрешено обращаться
 * к серверу класса Server. */
request dst=Server { grant () }

/* Клиенту класса Client запрещено
 * обращаться к серверу класса Server. */
request src=Client dst=Server { deny () }

/* Клиенту класса Client разрешено
 * обращаться к серверу класса Server,
 * вызывая метод Ping службы net.Net. */
request src=Client dst=Server endpoint=net.Net method=Ping {
    grant ()
}

/* Любому клиенту в решении разрешено обращаться
 * к серверу класса Server, вызывая метод Send
 * службы с интерфейсом MessExch. */
request dst=Server interface=MessExch method=Send {
    grant ()
}
```

Обработка отправки IPC-ответов

```
/* Серверу класса Server разрешено отвечать на
 * обращения клиента класса Client, который
 * вызывает метод Ping службы net.Net. */
response src=Server, dst=Client, endpoint=net.Net, method=Ping {
    grant ()
}

/* Серверу, который содержит компонент kl.drivers.KIDF,
 * предоставляющий службы с интерфейсом monitor, разрешено
 * отвечать на обращения клиента класса DriverManager,
 * который использует эти службы. */
response dst=DriverManager component=kl.drivers.KIDF interface=monitor {
    grant ()
}
```

Обработка отправки IPC-ответов, содержащих сведения об ошибках

```
/* Серверу класса Server запрещено сообщать клиенту
 * класса Client об ошибках, которые возникают,
 * когда клиент обращается к серверу, вызывая метод
 * Ping службы net.Net. */
error src=Server, dst=Client, endpoint=net.Net, method=Ping {
    deny ()
}
```

Обработка обращений процессов к модулю безопасности Kaspersky Security Module

```
/* Процесс класса Sdcard получит решение
 * "разрешено" от модуля безопасности Kaspersky Security Module,
 * вызывая метод Register интерфейса безопасности.
 * (Используется интерфейс безопасности, заданный
 * в EDL-описании.) */
security src=Sdcard, method=Register {
    grant ()
}

/* Процесс класса Sdcard получит решение "запрещено"
 * от модуля безопасности, вызывая метод Comp.Register
 * интерфейса безопасности. (Используется интерфейс
 * безопасности, заданный в CDL-описании.) */
security src=Sdcard, method=Comp.Register {
    deny ()
}
```

Использование match-секций

```
/* Клиенту класса Client разрешено обращаться к
 * серверу класса Server, вызывая методы Send
 * и Receive службы net. */
request src=Client, dst=Server, endpoint=net {
    match method=Send { grant () }
    match method=Receive { grant () }
}

/* Клиенту класса Client разрешено обращаться к
 * серверу класса Server, вызывая методы Send
 * и Receive службы sn.Net и методы Write и
 * Read службы sn.Storage. */
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 () }
    }
}
```

Установка профилей аудита

```
/* Установка глобального профиля аудита default
 * и начального уровня аудита 0 */
audit default = global 0
request src=Client, dst=Server {
    /* Установка профиля аудита parent на уровне
     * привязки методов моделей безопасности к
     * событиям безопасности */
    audit parent
    match endpoint=net.Net, method=Send {
        /* Установка профиля аудита child на
         * на уровне match-секции */
        audit child
        grant ()
        }
    /* В этой match-секции применяется профиль
     * аудита parent. */
    match endpoint=net.Net, method=Receive {
        grant ()
        }
}
/* В этой привязке метода модели безопасности
 * к событию безопасности применяется профиль
 * аудита global. */
response src=Client, dst=Server {
     grant ()
}
```

Примеры описаний простейших политик безопасности решений на базе KasperskyOS

> Перед тем как рассматривать примеры, нужно ознакомиться со сведениями о моделях безопасности Struct, Base и Flow.

Пример 1

Политика безопасности решения в этом примере разрешает любые взаимодействия процессов классов `Client`, `Server` и `Einit` между собой и с ядром KasperskyOS. При обращении процессов к модулю безопасности Kaspersky Security Module всегда будет получено решение "разрешено". Эту политику можно использовать только в качестве заглушки на ранних стадиях разработки решения на базе KasperskyOS, чтобы модуль безопасности Kaspersky Security Module "не мешал" взаимодействиям. В реальном решении на базе KasperskyOS применять такую политику недопустимо.

`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 () }
```

Пример 2

Политика безопасности решения в этом примере накладывает ограничения на обращения клиентов класса `FsClient` к серверам класса `FsDriver`. Когда клиент открывает ресурс, управляемый сервером класса `FsDriver`, с этим ресурсом ассоциируется конечный автомат в состоянии `unverified`. Клиенту класса `FsClient` разрешено читать данные из ресурса, управляемого сервером класса `FsDriver`, только если конечный автомат, ассоциированный с этим ресурсом, находится в состоянии `verified`. Чтобы перевести конечный автомат, ассоциированный с ресурсом, из состояния `unverified` в состояние `verified`, процессу класса `FsVerifier` нужно обратиться к модулю безопасности Kaspersky Security Module.

В реальном решении на базе KasperskyOS эту политику применять нельзя, поскольку разрешено избыточное множество взаимодействий процессов между собой и с ядром KasperskyOS.

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"}
}
```

Примеры профилей аудита безопасности

> Перед тем как рассматривать примеры, нужно ознакомиться со сведениями о моделях безопасности Base, Regex и Flow.

Пример 1

```
// Описание профиля аудита безопасности trace
// base – объект модели безопасности Base
// session – объект модели безопасности Flow
audit profile trace =
/* Если уровень аудита равен 0, аудитом покрываются
 * правила объекта base, когда эти правила возвращают
 * результат "запрещено". */
  { 0 :
      { base :
          { kss : ["denied"]
          }
      }
/* Если уровень аудита равен 1, аудитом покрываются методы
 * объекта session в следующих случаях:
 * 1. Правила объекта session возвращают любой результат, и
 * конечный автомат находится в состоянии, отличном от closed.
 * 2. Выражение query объекта session выполняется, и конечный
 * автомат находится в состоянии, отличном от closed. */
  , 1 :
      { session :
          { kss : ["granted", "denied"]
          , omit : ["closed"]
          }
      }
/* Если уровень аудита равен 2, аудитом покрываются методы
 * объекта session в следующих случаях:
 * 1. Правила объекта session возвращают любой результат.
 * 2. Выражение query объекта session выполняется. */
  , 2 :
      { session :
          { kss : ["granted", "denied"]
          }
      }
```

```
}
```

Пример 2

```
// Описание профиля аудита безопасности test
// base – объект модели безопасности Base
// re – объект модели безопасности Regex
audit profile test =
/* Если уровень аудита равен 0, правила объекта base
 * и выражения объекта re не покрываются аудитом. */
  { 0 :
      { base :
        { kss : []
        }
      , re :
        { kss : []
        , emit : []
        }
      }
/* Если уровень аудита равен 1, правила объекта
 * base не покрываются аудитом, выражения объекта
 * re покрываются аудитом.*/
  , 1 :
      { base :
        { kss : []
        }
      , re :
        { kss : ["granted"]
        , emit : ["match", "select"]
        }
      }
/* Если уровень аудита равен 2, правила объекта base
 * и выражения объекта re покрываются аудитом. Правила
 * объекта base покрываются аудитом независимо от
 * результата, который они возвращают.*/
  , 2 :
      { base :
        { kss : ["granted", "denied"]
        }
      , re :
        { kss : ["granted"]
        , emit : ["match", "select"]
        }
      }
  }
```

Примеры тестов политик безопасности решений на базе KasperskyOS

Пример 1

```
/* Набор тестов, который включает один тест. */
assert "some tests" {
    /* Тест, который включает четыре тестовых примера. */
    sequence "first sequence" {
        /* Ожидается, что запуск процесса класса Server разрешен.
         * Если это так, переменной s будет присвоено значение SID
         * запущенного процесса класса Server. */
        s <- execute dst=Server
        /* Ожидается, что запуск процесса класса Client разрешен.
         * Если это так, переменной c будет присвоено значение SID
         * запущенного процесса класса Client. */
        c <- execute dst=Client
        /* Ожидается, что клиенту класса Client разрешено обращаться к
         * серверу класса Server, вызывая метод Ping службы pingComp.pingImpl
         * с параметром value, равным 100. */
        grant "Client calls Ping" request src=c dst=s endpoint=pingComp.pingImpl
                                          method=Ping { value : 100 }
        /* Ожидается, что серверу класса Server запрещено отвечать клиенту
         * класса Client, если клиент вызывает метод Ping службы pingComp.pingImpl.
         * (IPC-ответ не содержит параметров, так как интерфейсный метод Ping
         * не имеет выходных параметров.) */
        deny "Server can not respond" response src=s dst=c endpoint=pingComp.pingImpl
                                                  method=Ping {}
    }
}
```

Пример 2

```
/* Набор тестов, который включает два теста. */
assert "ping tests"{
    /* Начальная часть каждого из двух тестов,
     * которая включает два тестовых примера. */
    setup {
        /* Ожидается, что запуск процесса класса Server разрешен.
         * Если это так, переменной s будет присвоено значение SID
         * запущенного процесса класса Server. */
        s <- execute dst=Server
        /* Ожидается, что запуск процесса класса Client разрешен.
         * Если это так, переменной c будет присвоено значение SID
         * запущенного процесса класса Client. */
        c <- execute dst=Client
    }
    /* Тест, который включает четыре тестовых примера: два тестовых примера
     * в начальной части и два тестовых примера в основной части.*/
    sequence "ping-ping is denied" {
        /* Ожидается, что клиенту класса Client разрешено обращаться к
         * серверу класса Server, вызывая метод Ping службы pingComp.pingImpl
         * с параметром value, равным 100. */
        c ~> s : pingComp.pingImpl.Ping { value : 100 }
        /* Ожидается, что клиенту класса Client запрещено обращаться к
         * серверу класса Server, повторно вызывая метод Ping службы pingComp.pingImpl
         * с параметром value, равным 100. */
        deny c ~> s : pingComp.pingImpl.Ping { value : 100 }
    }
    /* Тест, который включает четыре тестовых примера: два тестовых примера
     * в начальной части и два тестовых примера в основной части. */
    sequence "ping-pong is granted" {
        /* Ожидается, что клиенту класса Client разрешено обращаться к
```

```
         * серверу класса Server, вызывая метод Ping службы pingComp.pingImpl
         * с параметром value, равным 100. */
        c ~> s : pingComp.pingImpl.Ping { value: 100 }
        /* Ожидается, что клиенту класса Client разрешено обращаться к
         * серверу класса Server, вызывая метод Pong службы pingComp.pingImpl
         * с параметром value, равным 100. */
        c ~> s : pingComp.pingImpl.Pong { value: 100 }
    }
}
```

Пример 3

```
/* Набор тестов, который включает один тест. */
assert {
    /* Тест, который включает восемь тестовых примеров. */
    sequence {
        storage <- execute dst=test.kl.UpdateStorage
        manager <- execute dst=test.kl.UpdateManager
        deployer <- execute dst=test.kl.UpdateDeployer
        downloader <- execute dst=test.kl.UpdateDownloader
        grant manager ~>
              downloader:UpdateDownloader.Downloader.LoadPackage { url :
”url012345678” }
        grant response src=downloader dst=manager endpoint=UpdateDownloader.Downloader
                      method=LoadPackage { handle : 29, result : 1 }
        deny manager ~> deployer:UpdateDeployer.Deployer.Start { handle : 29 }
        deny request src=manager dst=deployer endpoint=UpdateDeployer.Deployer
                     method=Start { handle : 29 }
    }
}
```

Модели безопасности KasperskyOS

Модель безопасности Pred

Модель безопасности Pred позволяет выполнять операции сравнения.

PSL-файл с описанием модели безопасности Pred находится в KasperskyOS SDK по пути:

`toolchain/include/nk/basic.psl`

Объект модели безопасности Pred

В файле `basic.psl` содержится декларация, которая создает объект модели безопасности Pred с именем `pred`. Соответственно, включение файла `basic.psl` в описание политики безопасности решения обеспечивает создание объекта модели безопасности Pred по умолчанию.

Объект модели безопасности Pred не имеет параметров и не может быть покрыт аудитом безопасности.

Создавать дополнительные объекты модели безопасности Pred не требуется.

Методы модели безопасности Pred

Модель безопасности Pred содержит выражения, которые выполняют операции сравнения и возвращают значения типа `Boolean`. Для вызова этих выражений нужно использовать следующие операторы сравнения:

- `<ScalarLiteral> == <ScalarLiteral>` – "равно";
- `<ScalarLiteral> != <ScalarLiteral>` – "не равно";
- `<Number> < <Number>` – "меньше";
- `<Number> <= <Number>` – "меньше или равно";
- `<Number> > <Number>` – "больше";
- `<Number> >= <Number>` – "больше или равно".

Также модель безопасности Pred содержит выражение `empty`, которое позволяет определить, содержат ли данные свои структурные элементы. Выражение возвращает значения типа `Boolean`. Если данные не содержат своих структурных элементов (например, множество является пустым), выражение возвращает `true`, иначе возвращает `false`. Чтобы вызвать выражение, нужно использовать следующую конструкцию:

```
pred.empty (<Text | Set | List | Map>) | ()
```

Модель безопасности Bool

Модель безопасности Bool позволяет выполнять логические операции.

PSL-файл с описанием модели безопасности Bool находится в KasperskyOS SDK по пути:

`toolchain/include/nk/basic.psl`

Объект модели безопасности Bool

В файле `basic.psl` содержится декларация, которая создает объект модели безопасности Bool с именем `bool`. Соответственно, включение файла `basic.psl` в описание политики безопасности решения обеспечивает создание объекта модели безопасности Bool по умолчанию.

Объект модели безопасности Bool не имеет параметров и не может быть покрыт аудитом безопасности.

Создавать дополнительные объекты модели безопасности Bool не требуется.

Методы модели безопасности Bool

Модель безопасности Bool содержит выражения, которые выполняют логические операции и возвращают значения типа `Boolean`. Для вызова этих выражений нужно использовать следующие логические операторы:

- `! <Boolean>` – "логическое НЕ";
- `<Boolean> && <Boolean>` – "логическое И";
- `<Boolean> || <Boolean>` – "логическое ИЛИ";
- `<Boolean> ==> <Boolean>` – "импликация" (`! <Boolean> || <Boolean>`).

Также модель безопасности Bool содержит выражения `all`, `any` и `cond`.

Выражение `all` выполняет "логическое И" для произвольного числа значений типа `Boolean`. Возвращает значения типа `Boolean`. Если передать через параметр пустой список значений (`[]`), возвращает `true`. Чтобы вызвать выражение, нужно использовать следующую конструкцию:

```
bool.all (<List<Boolean>>)
```

Выражение `any` выполняет "логическое ИЛИ" для произвольного числа значений типа `Boolean`. Возвращает значения типа `Boolean`. Если передать через параметр пустой список значений (`[]`), возвращает `false`. Чтобы вызвать выражение, нужно использовать следующую конструкцию:

```
bool.any (<List<Boolean>>)
```

Выражение `cond` выполняет тернарную условную операцию. Возвращает значения типа `ScalarLiteral`. Чтобы вызвать выражение, нужно использовать следующую конструкцию:

```
bool.cond
  { if   : <Boolean> // Условие
  , then : <ScalarLiteral> // Значение, возвращаемое при истинности условия
  , else : <ScalarLiteral> // Значение, возвращаемое при ложности условия
  }
```

Помимо выражений модель безопасности Bool включает правило `assert`, которое работает так же, как одноименное правило модели безопасности Base.

Модель безопасности Math

Модель безопасности Math позволяет выполнять операции целочисленной арифметики.

PSL-файл с описанием модели безопасности Math находится в KasperskyOS SDK по пути:

`toolchain/include/nk/basic.psl`

Объект модели безопасности Math

В файле `basic.psl` содержится декларация, которая создает объект модели безопасности Math с именем `math`. Соответственно, включение файла `basic.psl` в описание политики безопасности решения обеспечивает создание объекта модели безопасности Math по умолчанию.

Объект модели безопасности Math не имеет параметров и не может быть покрыт аудитом безопасности.

Создавать дополнительные объекты модели безопасности Math не требуется.

Методы модели безопасности Math

Модель безопасности Math содержит выражения, которые выполняют операции целочисленной арифметики. Для вызова части этих выражений нужно использовать следующие арифметические операторы:

- `<Number> + <Number>` – "сложение". Возвращает значения типа `Number`.
- `<Number> - <Number>` – "вычитание". Возвращает значения типа `Number`.
- `<Number> * <Number>` – "умножение". Возвращает значения типа `Number`.

Другая часть включает следующие выражения:

- `neg (<Signed>)` – "изменение знака числа". Возвращает значения типа `Signed`.
- `abs (<Signed> )` – "получение модуля числа". Возвращает значения типа `Signed`.
- `sum (<List<Number>>)` – "сложение чисел из списка". Возвращает значения типа `Number`. Если передать через параметр пустой список значений (`[]`), возвращает `0`.
- `product (<List<Number>>)` – "перемножение чисел из списка". Возвращает значения типа `Number`. Если передать через параметр пустой список значений (`[]`), возвращает `1`.

Для вызова этих выражений нужно использовать следующую конструкцию:

```
math.<имя выражения> (<параметр>)
```

Модель безопасности Struct

Модель безопасности Struct позволяет получать доступ к структурным элементам данных.

PSL-файл с описанием модели безопасности Struct находится в KasperskyOS SDK по пути:

`toolchain/include/nk/basic.psl`

Объект модели безопасности Struct

В файле `basic.psl` содержится декларация, которая создает объект модели безопасности Struct с именем `struct`. Соответственно, включение файла `basic.psl` в описание политики безопасности решения обеспечивает создание объекта модели безопасности Struct по умолчанию.

Объект модели безопасности Struct не имеет параметров и не может быть покрыт аудитом безопасности.

Создавать дополнительные объекты модели безопасности Struct не требуется.

Методы модели безопасности Struct

Модель безопасности Struct содержит выражения, которые обеспечивают доступ к структурным элементам данных. Для вызова этих выражений нужно использовать следующие конструкции:

- `<словарь>.<имя поля>` – "получение доступа к полю словаря". Тип возвращаемых данных соответствует типу поля словаря.
- `<List | Set | Sequence | Array>.[<номер элемента>]` – "получение доступа к элементу данных". Тип возвращаемых данных соответствует типу элементов. Нумерация элементов начинается с нуля. При выходе за границы набора данных выражение завершается некорректно, и модуль безопасности Kaspersky Security Module возвращает решение "запрещено".
- `<HandleDesc>.handle` – "получение SID". Возвращает значения типа `Handle`. (О взаимосвязи между дескрипторами и значениями SID см. "Управление доступом к ресурсам").
- `<HandleDesc>.rights` – "получение маски прав дескриптора". Возвращает значения типа `UInt32`.

Параметры интерфейсных методов сохраняются в специальном словаре `message`. Чтобы получить доступ к параметру интерфейсного метода, нужно использовать следующую конструкцию:

```
message.<имя параметра интерфейсного метода>
```

Имя параметра нужно указать в соответствии с IDL-описанием.

Чтобы получить доступ к структурным элементам параметров, нужно использовать конструкции, соответствующие выражениям модели безопасности Struct.

> Чтобы использовать выражения модели безопасности Struct, описание события безопасности должно быть настолько точным, чтобы ему соответствовали IPC-сообщения одного типа (подробнее см. "Привязка методов моделей безопасности к событиям безопасности"). IPC-сообщения этого типа должны содержать заданные параметры интерфейсного метода, и параметры интерфейсного метода должны содержать заданные структурные элементы.

Модель безопасности Base

Модель безопасности Base позволяет реализовать простейшую логику.

PSL-файл с описанием модели безопасности Base находится в KasperskyOS SDK по пути:

`toolchain/include/nk/base.psl`

Объект модели безопасности Base

В файле `base.psl` содержится декларация, которая создает объект модели безопасности Base с именем `base`. Соответственно, включение файла `base.psl` в описание политики безопасности решения обеспечивает создание объекта модели безопасности Base по умолчанию. Методы этого объекта можно вызывать без указания имени объекта.

Объект модели безопасности Base не имеет параметров.

Объект модели безопасности Base может быть покрыт аудитом безопасности. Условия выполнения аудита, специфичные для модели безопасности Base, отсутствуют.

Необходимость создавать дополнительные объекты модели безопасности Base возникает в следующих случаях:

- Если нужно по-разному настроить аудит безопасности для разных объектов модели безопасности Base (например, для разных объектов можно применять разные профили аудита или разные конфигурации аудита одного профиля).

- Если нужно различать вызовы методов, предоставляемых разными объектами модели безопасности Base (поскольку в данные аудита включается как имя метода модели безопасности, так и имя объекта, предоставляющего этот метод, можно понять, что был вызван метод конкретного объекта).

Методы модели безопасности Base

Модель безопасности Base содержит следующие правила:

- `grant ()`

  Имеет параметр типа `()`. Возвращает результат "разрешено".

  Пример:

  ```
  /* Клиенту класса foo разрешено
   * обращаться к серверу класса bar. */
  request src=foo dst=bar { grant () }
  ```

- `assert (<Boolean>)`

  Возвращает результат "разрешено", если через параметр передать значение `true`. Иначе возвращает результат "запрещено".

  Пример:

  ```
  /* Любому клиенту в решении будет разрешено обращаться к серверу класса
   * foo, вызывая метод Send службы net.Net, если через параметр port
   * метода Send будет передаваться значение больше 80. Иначе любому
   * клиенту в решении будет запрещено обращаться к серверу класса
   * foo, вызывая метод Send службы net.Net. */
  request dst=foo endpoint=net.Net method=Send { assert (message.port > 80) }
  ```

- `deny (<Boolean>) | ()`

  Возвращает результат "запрещено", если через параметр передать значение `true` или `()`. Иначе возвращает результат "разрешено".

  Пример:

  ```
  /* Серверу класса foo запрещено
   * отвечать клиенту класса bar. */
  response src=foo dst=bar { deny () }
  ```

- `set_level (<UInt8>)`

Устанавливает уровень аудита безопасности равным значению, переданному через параметр. Возвращает результат "разрешено". (Подробнее об уровне аудита безопасности см. "Описание профилей аудита безопасности".)

Пример:

```
/* Процесс класса foo получит решение "разрешено" от модуля
 * безопасности Kaspersky Security Module, если вызовет метод интерфейса
безопасности
 * SetAuditLevel, чтобы изменить уровень аудита безопасности. */
security src=foo method=SetAuditLevel { set_level (message.audit_level) }
```

Модель безопасности Regex

Модель безопасности Regex позволяет реализовать валидацию текстовых данных по статически заданным регулярным выражениям.

PSL-файл с описанием модели безопасности Regex находится в KasperskyOS SDK по пути:

`toolchain/include/nk/regex.psl`

Объект модели безопасности Regex

В файле `regex.psl` содержится декларация, которая создает объект модели безопасности Regex с именем `re`. Соответственно, включение файла `regex.psl` в описание политики безопасности решения обеспечивает создание объекта модели безопасности Regex по умолчанию.

Объект модели безопасности Regex не имеет параметров.

Объект модели безопасности Regex может быть покрыт аудитом безопасности. При этом нужно задать условия выполнения аудита, специфичные для модели безопасности Regex. Для этого в описании конфигурации аудита нужно использовать следующие конструкции:

- `emit : ["match"]` – аудит выполняется, если вызван метод `match`;
- `emit : ["select"]` – аудит выполняется, если вызван метод `select`;
- `emit : ["match", "select"]` – аудит выполняется, если вызван метод `match` или `select`;
- `emit : []` – аудит не выполняется.

Необходимость создавать дополнительные объекты модели безопасности Regex возникает в следующих случаях:

- Если нужно по-разному настроить аудит безопасности для разных объектов модели безопасности Regex (например, для разных объектов можно применять разные профили аудита или разные конфигурации аудита одного профиля).
- Если нужно различать вызовы методов, предоставляемых разными объектами модели безопасности Regex (поскольку в данные аудита включается как имя метода модели безопасности, так и имя объекта, предоставляющего этот метод, можно понять, что был вызван метод конкретного объекта).

Методы модели безопасности Regex

Модель безопасности `Regex` содержит следующие выражения:

- `match {text : <Text>, pattern : <Text>}`

  Возвращает значение типа `Boolen`. Если текст `text` соответствует регулярному выражению `pattern`, возвращает `true`. Иначе возвращает `false`.

  Пример:

  ```
  assert (re.match {text : message.text, pattern : "[0-9]*"})
  ```

- `select {text : <Text>}`

  Предназначено для использования в качестве выражения, проверяющего выполнение условий в конструкции `choice` (о конструкции `choice` см. "Привязка методов моделей безопасности к событиям безопасности"). Проверяет соответствие текста `text` регулярным выражениям. В зависимости от результатов этой проверки выполняются различные варианты обработки события безопасности.

  Пример:

  ```
  choice (re.select {text : "hello world"}) {
      "hello\ .*": grant ()
      ".*world" : grant ()
      _         : deny ()
    }
  ```

Синтаксис регулярных выражений модели безопасности Regex

Регулярное выражение для метода `match` модели безопасности Regex можно записать двумя способами: внутри многострочного блока `regex` или как текстовый литерал.

> При записи регулярного выражения как текстового литерала все вхождения обратного слеша необходимо удвоить.

Например, два регулярных выражения идентичны:

```
// Регулярное выражение внутри многострочного блока regex
{ pattern:
    ```regex
    Hello\ world\!
    ```
, text: "Hello world!"
}
// Регулярное выражение как текстовый литерал (обратный слеш удвоен)
{ pattern: "Hello\\ world\\!"
, text: "Hello world!"
}
```

Регулярные выражения для метода `select` модели безопасности Regex записываются как текстовые литералы с удвоением обратного слеша.

Регулярное выражение задается в виде строки-шаблона и может содержать:

- литералы (обычные символы);
- метасимволы (символы со специальными значениями);
- пробельные символы;
- наборы символов;
- группы символов;
- операторы для работы с символами.

Регулярные выражения чувствительны к регистру.

Литералы и метасимволы в регулярных выражениях

- Литералом является любой ASCII-символ, за исключением метасимволов `.()*&|!?+[]\` и знака пробела. (Символы Unicode не поддерживаются.)

  Например, регулярному выражению `KasperskyOS` соответствует текст `KasperskyOS`.

- Метасимволы имеют специальные значения, которые приведены в таблице ниже.

Специальные значения метасимволов

Метасимвол	Специальное значение
`[]`	Квадратные скобки обозначают начало и конец набора символов.
`()`	Круглые скобки обозначают начало и конец группы символов.
`*`	Звездочка обозначает оператор, определяющий, что символ, который ему предшествует, может повторяться ноль или больше раз.
`+`	Плюс обозначает оператор, определяющий, что символ, который ему предшествует, может повторяться один или больше раз.
`?`	Вопрос обозначает оператор, определяющий, что символ, который ему предшествует, может повторяться ноль или один раз.
`!`	Восклицательный знак обозначает оператор, исключающий последующий символ из списка допустимых символов.
`|`	Вертикальная черта обозначает оператор выбора между символами (близок по смыслу к союзу "ИЛИ").
`&`	Амперсанд обозначает оператор пересечения нескольких условий (близок по смыслу к союзу "И").
`.`	Точка обозначает соответствие любому символу. Например, регулярному выражению `K.S` соответствуют последовательности символов: `KOS`, `KoS`, `KES` и множество других последовательностей из трех символов, которые начинаются на `K` и заканчиваются на `S`, и где второй символ может быть любым: литералом, метасимволом или самой точкой.
`\`	`\<metaSymbol>` Обратный слеш указывает, что следующий за ним метасимвол будет интерпретирован как литерал и потеряет свое специальное значение. Добавление обратного слеша перед метасимволом называется экранированием метасимвола.

	Например, регулярному выражению, которое состоит из метасимвола точки `.`, соответствует любой символ, а регулярному выражению, которое состоит из обратного слеша с точкой `\.`, соответствует только сам символ точки. Аналогично обратный слеш действует и на самого себя. Регулярному выражению `C:\\Users` соответствует последовательность символов `C:\Users`.

- Символы `^` и `$` как обозначения начала и конца строки не используются.

Пробельные символы в регулярных выражениях

- Знак пробела имеет ASCII-код, равный `20`, в шестнадцатеричной системе счисления и ASCII-код, равный `40`, в восьмеричной системе счисления. Знак пробела не имеет специального значения, но во избежание неоднозначной трактовки данного символа интерпретатором регулярных выражений, пробел необходимо экранировать.

  Например, регулярному выражению `Hello\ world` соответствует последовательность символов `Hello world`.

- `\r`

  Символ возврата каретки.

- `\n`

  Символ переноса строки.

- `\t`

  Символ горизонтальной табуляции.

Определение символа по его восьмеричному или шестнадцатеричному коду в регулярных выражениях

- `\x{<hex>}`

  Определение символа его шестнадцатеричным кодом `hex` из таблицы ASCII-символов. Код символа должен быть меньше, чем `0x100`.

  Например, регулярному выражению `Hello\x{20}world` соответствует последовательность символов `Hello world`.

- `\o{<octal>}`

  Определение символа его восьмеричным кодом `octal` из таблицы ASCII-символов. Код символа должен быть меньше, чем `0o400`.

  Например, регулярному выражению `\o{75}` соответствует символ `=`.

Наборы символов в регулярных выражениях

Набор символов задается внутри квадратных скобок `[]` перечислением или диапазоном символов. Набор символов указывает интерпретатору регулярных выражений, что на этом месте в последовательности символов может стоять только один из перечисленных в наборе или диапазоне символов. Набор символов не может быть пустым.

- `[<BracketSpec>]` – набор символов.

  Один символ соответствует любому символу из набора символов `BracketSpec`.

Например, регулярному выражению `K[OE]S` соответствуют последовательности символов `KOS` и `KES`.

- `[^<BracketSpec>]` – набор символов с инверсией.

  Один символ соответствует любому символу, не входящему в набор символов `BracketSpec`.

  Например, регулярному выражению `K[^OE]S` соответствуют последовательности символов `KAS`, `K8S` и любые другие последовательности из трех символов, которые начинаются на `K` и заканчиваются на `S`, кроме `KOS` и `KES`.

Набор символов `BracketSpec` может быть перечислен явно или определен как диапазон символов. Чтобы определить диапазон символов, первый и последний символ в наборе разделяют дефисом.

- `[<Digit1>-<DigitN>]`

  Любая цифра из диапазона `Digit1`, `Digit2`, … ,`DigitN`.

  Например, регулярному выражению `[0-9]` соответствует любая цифра. Записи регулярных выражений `[0-9]` и `[0123456789]` идентичны.

  Обратите внимание, что диапазон определяется одним символом до и одним символом после дефиса. Регулярному выражению `[1-35]` соответствуют символы `1`, `2`, `3` и `5`, а не диапазон чисел от `1` до `35`.

- `[<Letter1>-<LetterN>]`

  Любая латинская буква из диапазона `Letter1`, `Letter2`, … , `LetterN` (буквы должны быть в одинаковых регистрах).

  Например, регулярному выражению `[a-zA-Z]` соответствуют все буквы в нижнем и верхнем регистре из таблицы ASCII-символов.

ASCII-код символа верхней границы диапазона должен быть больше ASCII-кода символа нижней границы диапазона.

Например, регулярные выражения типа `[5-2]` или `[z-a]` недопустимы.

Знак дефиса (минуса) `-` рассматривается как специальный символ только внутри набора символов. Вне набора символов дефис является литералом, поэтому дефису не обязан предшествовать метасимвол `\`. Для использования дефиса как литерала внутри набора символов необходимо указывать его первым или последним в наборе.

Примеры:

Регулярным выражениям `[-az]` и `[az-]` соответствуют символы `a`, `z` и `-`.

Регулярному выражению `[a-z]` соответствует любая из 26 латинских букв от `a` до `z` в нижнем регистре.

Регулярному выражению `[-a-z]` соответствует любая из 26 латинских букв от `a` до `z` в нижнем регистре и `-`.

Циркумфлекс (символ вставки) `^` рассматривается как специальный символ только внутри набора символов, когда он расположен сразу после открывающей квадратной скобки. Вне набора символов циркумфлекс является литералом, поэтому циркумфлексу не обязан предшествовать метасимвол `\`. Для использования циркумфлекса как литерала внутри набора символов необходимо указывать его не первым в наборе.

Примеры:

Регулярному выражению `[0^9]` соответствуют символы `0`, `9` и `^`.

Регулярному выражению `[^09]` соответствует любой символ, кроме `0` и `9`.

Внутри набора символов метасимволы `*.&|!?+` теряют свое специальное значение и интерпретируются как литералы, поэтому предварять их метасимволом `\` не обязательно. Обратный слеш `\` сохраняет свое специальное значение внутри набора символов.

Например, регулярные выражения `[a.]` и `[a\.]` идентичны, и им соответствуют символ `a` и точка как литерал.

Группы символов и операторы в регулярных выражениях

Группа символов выделяет из регулярного выражения его часть (подвыражение) с помощью круглых скобок `()`. Обычно группы используются для выделения подвыражений в качестве операндов. Группы могут быть вложены друг в друга.

> Операторы применяются более чем к одному символу в регулярном выражении, только если они стоят сразу перед или после определения набора или группы символов. В этом случае действие оператора распространяется на всю группу или набор символов.

В синтаксисе определены следующие операторы (перечислены в порядке убывания приоритета):

- `!<Expression>`, где `Expression` может быть символом, набором или группой символов.

  Оператор означает исключение выражения `Expression` из списка допустимых выражений.

  Примеры:

  Регулярному выражению `K!OS` соответствуют последовательности символов `KoS`, `KES` и множество других последовательностей, которые состоят из трех символов, начинаются на `K` и заканчиваются на `S`, кроме `KOS`.

  Регулярному выражению `K!(OS)` соответствуют последовательности символов `Kos`, `KES`, `KOT` и множество других последовательностей, которые состоят из трех символов и начинаются на `K`, кроме `KOS`.

  Регулярному выражению `K![OE]S` соответствуют последовательности символов `KoS`, `KeS`, `K;S` и множество других последовательностей, которые состоят из трех символов, начинаются на `K` и заканчиваются на `S`, кроме `KOS` и `KES`.

- `<Expression>*`, где `Expression` может быть символом, набором или группой символов.

  Оператор означает, что выражение `Expression` может встретиться в этой позиции ноль или больше раз.

  Примеры:

  Регулярному выражению `0-9*` соответствуют последовательности символов `0-`, `0-9`, `0-99`, … .

  Регулярному выражению `(0-9)*` соответствуют пустая последовательность `""` и последовательности символов `0-9`, `0-90-9`, … .

  Регулярному выражению `[0-9]*` соответствуют пустая последовательность `""` и любая непустая последовательность цифр.

- `<Expression>+`, где `Expression` может быть символом, набором или группой символов.

  Оператор означает, что выражение `Expression` может встретиться в этой позиции один или больше раз.

  Примеры:

  Регулярному выражению `0-9+` соответствуют последовательности символов `0-9`, `0-99`, `0-999`, … .

  Регулярному выражению `(0-9)+` соответствуют последовательности символов `0-9`, `0-90-9`, … .

  Регулярному выражению `[0-9]+` соответствует любая непустая последовательность цифр.

- `<Expression>?`, где `Expression` может быть символом, набором или группой символов.

  Оператор означает, что выражение `Expression` может встретиться в данной позиции ноль или один раз.

  Примеры:

  Регулярному выражению `https?://` соответствуют последовательности символов `http://` и `https://`.

  Регулярному выражению `K(aspersky)?OS` соответствуют последовательности символов `KOS` и `KaspershyOS`.

- `<Expression1><Expression2>` – конкатенация. `Expression1` и `Expression2` могут быть символами, наборами или группами символов.

  Оператор не имеет обозначения. В результирующем выражении за выражением `Expression1` следует выражение `Expression2`.

  Например, результатом конкатенации последовательностей символов `микро` и `ядро` будет последовательность символов `микроядро`.

- `<Expression1>|<Expression2>` – дизъюнкция. `Expression1` и `Expression2` могут быть символами, наборами или группами символов.

  Оператор означает выбор выражения `Expression1` или выражения `Expression2`.

  Примеры:

  Регулярному выражению `KO|ES` соответствуют последовательности символов `KO` и `ES`, но не `KOS` и не `KES`, так как оператор конкатенации имеет приоритет выше, чем оператор дизъюнкции.

  Регулярному выражению `Press (OK|Cancel)` соответствуют последовательности символов `Press OK` или `Press Cancel`.

  Регулярному выражению `[0-9]|()` соответствуют цифры от `0` до `9` или пустая строка.

- `<Expression1>&<Expression2>` – конъюнкция. `Expression1` и `Expression2` могут быть символами, наборами или группами символов.

  Оператор означает пересечение результата выражения `Expression1` с результатом выражения `Expression2`.

  Примеры:

  Регулярному выражению `[0-9]&[^3]` соответствуют цифры от `0` до `9`, кроме `3`.

  Регулярному выражению `[a-zA-Z]&()` соответствуют все латинские буквы и пустая строка.

Модель безопасности HashSet

Модель безопасности HashSet позволяет ассоциировать с ресурсами одномерные таблицы уникальных значений одного типа, добавлять и удалять эти значения, а также проверять, входит ли заданное значение в таблицу. Например, можно ассоциировать процесс, в контексте которого выполняется сетевой сервер, с набором портов, который разрешено открывать этому серверу. Эту ассоциацию можно использовать, чтобы проверить, допустимо ли открытие порта, инициированное сервером.

PSL-файл с описанием модели безопасности HashSet находится в KasperskyOS SDK по пути:

`toolchain/include/nk/hashmap.psl`

Объект модели безопасности HashSet

Чтобы использовать модель безопасности HashSet, нужно создать объект (объекты) этой модели.

Объект модели безопасности HashSet содержит пул одномерных таблиц одинакового размера, предназначенных для хранения значений одного типа. Ресурс может быть ассоциирован только с одной таблицей из пула таблиц каждого объекта модели безопасности HashSet.

Объект модели безопасности HashSet имеет следующие параметры:

- `type Entry` – тип значений в таблицах (поддерживаются целочисленные типы, тип `Boolean`, а также словари и кортежи на базе целочисленных типов и типа `Boolean`);
- `config` – конфигурация пула таблиц:
  - `set_size` – размер таблицы;
  - `pool_size` – число таблиц в пуле.

Все параметры объекта модели безопасности HashSet являются обязательными.

Пример:

```
policy object s : HashSet {
    type Entry = UInt32

    config =
      { set_size : 5
      , pool_size : 2
      }
}
```

Объект модели безопасности HashSet может быть покрыт аудитом безопасности. Условия выполнения аудита, специфичные для модели безопасности HashSet, отсутствуют.

Необходимость создавать несколько объектов модели безопасности HashSet возникает в следующих случаях:

- Если нужно по-разному настроить аудит безопасности для разных объектов модели безопасности HashSet (например, для разных объектов можно применять разные профили аудита или разные конфигурации аудита одного профиля).
- Если нужно различать вызовы методов, предоставляемых разными объектами модели безопасности HashSet (поскольку в данные аудита включается как имя метода модели безопасности, так и имя объекта, предоставляющего этот метод, можно понять, что был вызван метод конкретного объекта).
- Если нужно использовать таблицы разных размеров и/или с разными типами значений.

Правило init модели безопасности HashSet

```
init {sid : <Sid>}
```

Ассоциирует свободную таблицу из пула таблиц с ресурсом `sid`. Если свободная таблица содержит значения после предыдущего использования, то эти значения удаляются.

Возвращает результат "разрешено", если создало ассоциацию таблицы с ресурсом `sid`.

Возвращает результат "запрещено" в следующих случаях:

- В пуле нет свободных таблиц.
- Ресурс `sid` уже ассоциирован с таблицей из пула таблиц используемого объекта модели безопасности HashSet.
- Значение `sid` вне допустимого диапазона.

Пример:

```
/* Запуск процесса класса Server будет разрешен, если
 * при инициации запуска будет создана ассоциация этого
 * процесса с таблицей. Иначе запуск процесса класса
 * Server будет запрещен. */
execute dst=Server {
    s.init {sid : dst_sid}
}
```

Правило fini модели безопасности HashSet

```
fini {sid : <Sid>}
```

Удаляет ассоциацию таблицы с ресурсом `sid` (таблица становится свободной).

Возвращает результат "разрешено", если удалило ассоциацию таблицы с ресурсом `sid`.

Возвращает результат "запрещено" в следующих случаях:

- Ресурс `sid` не ассоциирован с таблицей из пула таблиц используемого объекта модели безопасности HashSet.
- Значение `sid` вне допустимого диапазона.

Правило add модели безопасности HashSet

```
add {sid : <Sid>, entry : <Entry>}
```

Добавляет значение `entry` в таблицу, ассоциированную с ресурсом `sid`.

Возвращает результат "разрешено" в следующих случаях:

- Правило добавило значение `entry` в таблицу, ассоциированную с ресурсом `sid`.
- В таблице, ассоциированной с ресурсом `sid`, уже содержится значение `entry`.

Возвращает результат "запрещено" в следующих случаях:

- Таблица, ассоциированная с ресурсом `sid`, полностью заполнена.
- Ресурс `sid` не ассоциирован с таблицей из пула таблиц используемого объекта модели безопасности HashSet.
- Значение `sid` вне допустимого диапазона.

Пример:

```
/* Процесс класса Server получит решение "разрешено" от
 * модуля безопасности Kaspersky Security Module, вызывая метод интерфейса
 * безопасности Add, если при вызове этого метода значение
 * 5 будет добавлено в таблицу, ассоциированную с этим
 * процессом, или уже содержится в этой таблице. Иначе
 * процесс класса Server получит решение "запрещено" от
 * модуля безопасности, вызывая метод интерфейса
 * безопасности Add. */
security src=Server, method=Add {
    s.add {sid : src_sid, entry : 5}
}
```

Правило remove модели безопасности HashSet

```
remove {sid : <Sid>, entry : <Entry>}
```

Удаляет значение `entry` из таблицы, ассоциированной с ресурсом `sid`.

Возвращает результат "разрешено" в следующих случаях:

- Правило удалило значение `entry` из таблицы, ассоциированной с ресурсом `sid`.
- В таблице, ассоциированной с ресурсом `sid`, нет значения `entry`.

Возвращает результат "запрещено" в следующих случаях:

- Ресурс `sid` не ассоциирован с таблицей из пула таблиц используемого объекта модели безопасности HashSet.
- Значение `sid` вне допустимого диапазона.

Выражение contains модели безопасности HashSet

```
contains {sid : <Sid>, entry : <Entry>}
```

Проверяет, содержится ли значение `entry` в таблице, ассоциированной с ресурсом `sid`.

Возвращает значение типа `Boolean`. Если значение `entry` содержится в таблице, ассоциированной с ресурсом `sid`, возвращает `true`. Иначе возвращает `false`.

Выполняется некорректно в следующих случаях:

- Ресурс `sid` не ассоциирован с таблицей из пула таблиц используемого объекта модели безопасности HashSet.
- Значение `sid` вне допустимого диапазона.

Когда выражение выполняется некорректно, модуль безопасности Kaspersky Security Module возвращает решение "запрещено".

Пример:

```
/* Процесс класса Server получит решение "разрешено" от
 * модуля безопасности Kaspersky Security Module, вызывая метод интерфейса
 * безопасности Check, если значение 42 содержится в таблице,
 * ассоциированной с этим процессом. Иначе процесс класса
 * Server получит решение "запрещено" от модуля безопасности,
 * вызывая метод интерфейса безопасности Check. */
security src=Server, method=Check {
    assert(s.contains {sid : src_sid, entry : 42})
}
```

Модель безопасности StaticMap

Модель безопасности StaticMap позволяет ассоциировать с ресурсами двумерные таблицы типа "ключ–значение", читать и изменять значения ключей. Например, можно ассоциировать процесс, в контексте которого выполняется драйвер, с регионом памяти MMIO, который разрешено использовать этому драйверу. Для этого потребуется два ключа, значения которых задают базовый адрес и размер региона памяти MMIO. Эту ассоциацию можно использовать, чтобы проверить, может ли драйвер обращаться к региону памяти MMIO, к которому он пытается получить доступ.

Ключи в таблице имеют одинаковый тип и являются уникальными и неизменяемыми. Значения ключей в таблице имеют одинаковый тип.

Одновременно существует два экземпляра таблицы: базовый и рабочий. Оба экземпляра инициализируются одинаковыми данными. Изменения заносятся сначала в рабочий экземпляр, а затем могут быть добавлены в базовый экземпляр или, наоборот, заменены прежними значениями из базового экземпляра. Значения ключей могут быть прочитаны как из базового, так и из рабочего экземпляра таблицы.

PSL-файл с описанием модели безопасности StaticMap находится в KasperskyOS SDK по пути:

`toolchain/include/nk/staticmap.psl`

Объект модели безопасности StaticMap

Чтобы использовать модель безопасности StaticMap, нужно создать объект (объекты) этой модели.

Объект модели безопасности StaticMap содержит пул двумерных таблиц типа "ключ–значение", которые имеют одинаковый размер. Ресурс может быть ассоциирован только с одной таблицей из пула таблиц каждого объекта модели безопасности StaticMap.

Объект модели безопасности StaticMap имеет следующие параметры:

- `type Value` – тип значений ключей в таблицах (поддерживаются целочисленные типы);
- `config` – конфигурация пула таблиц:
  - `keys` – таблица, содержащая ключи и их значения по умолчанию (ключи имеют тип `Key = Text | List<UInt8>`);
  - `pool_size` – число таблиц в пуле.

Все параметры объекта модели безопасности StaticMap являются обязательными.

Пример:

```
policy object m : StaticMap {
    type Value = UInt16

    config =
      { keys:
        { "k1" : 0
        , "k2" : 1
        }
      , pool_size : 2
      }
}
```

Объект модели безопасности StaticMap может быть покрыт аудитом безопасности. Условия выполнения аудита, специфичные для модели безопасности StaticMap, отсутствуют.

Необходимость создавать несколько объектов модели безопасности StaticMap возникает в следующих случаях:

- Если нужно по-разному настроить аудит безопасности для разных объектов модели безопасности StaticMap (например, для разных объектов можно применять разные профили аудита или разные конфигурации аудита одного профиля).
- Если нужно различать вызовы методов, предоставляемых разными объектами модели безопасности StaticMap (поскольку в данные аудита включается как имя метода модели безопасности, так и имя объекта, предоставляющего этот метод, можно понять, что был вызван метод конкретного объекта).
- Если нужно использовать таблицы с разными наборами ключей и/или разными типами значений ключей.

Правило init модели безопасности StaticMap

```
init {sid : <Sid>}
```

Ассоциирует свободную таблицу из пула таблиц с ресурсом `sid`. Ключи инициализируются значениями по умолчанию.

Возвращает результат "разрешено", если создало ассоциацию таблицы с ресурсом `sid`.

Возвращает результат "запрещено" в следующих случаях:

- В пуле нет свободных таблиц.
- Ресурс `sid` уже ассоциирован с таблицей из пула таблиц используемого объекта модели безопасности StaticMap.
- Значение `sid` вне допустимого диапазона.

Пример:

```
/* Запуск процесса класса Server будет разрешен, если
 * при инициации запуска будет создана ассоциация этого
 * процесса с таблицей. Иначе запуск процесса класса
 * Server будет запрещен. */
execute dst=Server {
    m.init {sid : dst_sid}
}
```

Правило fini модели безопасности StaticMap

```
fini {sid : <Sid>}
```

Удаляет ассоциацию таблицы с ресурсом `sid` (таблица становится свободной).

Возвращает результат "разрешено", если удалило ассоциацию таблицы с ресурсом `sid`.

Возвращает результат "запрещено" в следующих случаях:

- Ресурс `sid` не ассоциирован с таблицей из пула таблиц используемого объекта модели безопасности StaticMap.
- Значение `sid` вне допустимого диапазона.

Правило set модели безопасности StaticMap

```
set {sid : <Sid>, key : <Key>, value : <Value>}
```

Задает значение `value` ключу `key` в рабочем экземпляре таблицы, ассоциированной с ресурсом `sid`.

Возвращает результат "разрешено", если задало значение `value` ключу `key` в рабочем экземпляре таблицы, ассоциированной с ресурсом `sid`. (Текущее значение ключа будет перезаписано, даже если оно равно новому.)

Возвращает результат "запрещено" в следующих случаях:

- Ключ `key` не содержится в таблице, ассоциированной с ресурсом `sid`.
- Ресурс `sid` не ассоциирован с таблицей из пула таблиц используемого объекта модели безопасности StaticMap.
- Значение `sid` вне допустимого диапазона.

Пример:

```
/* Процесс класса Server получит решение "разрешено" от
 * модуля безопасности Kaspersky Security Module, вызывая метод интерфейса
 * безопасности Set, если при вызове этого метода значение 2
 * будет задано ключу k1 в рабочем экземпляре таблицы,
 * ассоциированной с этим процессом. Иначе процесс класса
 * Server получит решение "запрещено" от модуля безопасности,
 * вызывая метод интерфейса безопасности Set. */
security src=Server, method=Set {
    m.set {sid : src_sid, key : "k1", value : 2}
}
```

Правило commit модели безопасности StaticMap

```
commit {sid : <Sid>}
```

Копирует значения ключей из рабочего в базовый экземпляр таблицы, ассоциированной с ресурсом `sid`.

Возвращает результат "разрешено", если скопировало значения ключей из рабочего в базовый экземпляр таблицы, ассоциированной с ресурсом `sid`.

Возвращает результат "запрещено" в следующих случаях:

- Ресурс `sid` не ассоциирован с таблицей из пула таблиц используемого объекта модели безопасности StaticMap.
- Значение `sid` вне допустимого диапазона.

Правило rollback модели безопасности StaticMap

```
rollback {sid : <Sid>}
```

Копирует значения ключей из базового в рабочий экземпляр таблицы, ассоциированной с ресурсом `sid`.

Возвращает результат "разрешено", если скопировало значения ключей из базового в рабочий экземпляр таблицы, ассоциированной с ресурсом `sid`.

Возвращает результат "запрещено" в следующих случаях:

- Ресурс `sid` не ассоциирован с таблицей из пула таблиц используемого объекта модели безопасности StaticMap.
- Значение `sid` вне допустимого диапазона.

Выражение get модели безопасности StaticMap

```
get {sid : <Sid>, key : <Key>}
```

Возвращает значение ключа `key` из базового экземпляра таблицы, ассоциированной с ресурсом `sid`.

Возвращает значение типа `Value`.

Выполняется некорректно в следующих случаях:

- Ключ `key` не содержится в таблице, ассоциированной с ресурсом `sid`.
- Ресурс `sid` не ассоциирован с таблицей из пула таблиц используемого объекта модели безопасности StaticMap.
- Значение `sid` вне допустимого диапазона.

Когда выражение выполняется некорректно, модуль безопасности Kaspersky Security Module возвращает решение "запрещено".

Пример:

```
/* Процесс класса Server получит решение "разрешено" от
 * модуля безопасности Kaspersky Security Module, вызывая метод интерфейса
 * безопасности Get, если значение ключа k1 в базовом
 * экземпляре таблицы, ассоциированной с этим процессом,
 * отлично от нуля. Иначе процесс класса Server получит
 * решение "запрещено" от модуля безопасности, вызывая
 * метод интерфейса безопасности Get. */
security src=Server, method=Get {
    assert(m.get {sid : src_sid, key : "k1"} != 0)
}
```

Выражение get_uncommitted модели безопасности StaticMap

```
get_uncommitted {sid: <Sid>, key: <Key>}
```

Возвращает значение ключа `key` из рабочего экземпляра таблицы, ассоциированной с ресурсом `sid`.

Возвращает значение типа `Value`.

Выполняется некорректно в следующих случаях:

- Ключ `key` не содержится в таблице, ассоциированной с ресурсом `sid`.
- Ресурс `sid` не ассоциирован с таблицей из пула таблиц используемого объекта модели безопасности StaticMap.
- Значение `sid` вне допустимого диапазона.

Когда выражение выполняется некорректно, модуль безопасности Kaspersky Security Module возвращает решение "запрещено".

Модель безопасности Flow

Модель безопасности Flow позволяет ассоциировать с ресурсами конечные автоматы, получать и изменять состояния конечных автоматов, а также проверять, что состояние конечного автомата входит в заданный набор состояний. Например, можно ассоциировать процесс с конечным автоматом, чтобы разрешать и запрещать этому процессу использовать накопители и/или сеть в зависимости от состояния конечного автомата.

PSL-файл с описанием модели безопасности Flow находится в KasperskyOS SDK по пути:

`toolchain/include/nk/flow.psl`

Объект модели безопасности Flow

Чтобы использовать модель безопасности Flow, нужно создать объект (объекты) этой модели.

Один объект модели безопасности Flow позволяет ассоциировать множество ресурсов со множеством конечных автоматов, которые имеют одинаковую конфигурацию. Ресурс может быть ассоциирован только с одним конечным автоматом каждого объекта модели безопасности Flow.

Объект модели безопасности Flow имеет следующие параметры:

- `type State` – тип, определяющий множество состояний конечного автомата (вариантный тип, объединяющий текстовые литералы);
- `config` – конфигурация конечного автомата:
  - `states` – множество состояний конечного автомата (должно совпадать со множеством состояний, заданных типом `State`);
  - `initial` – начальное состояние конечного автомата;
  - `transitions` – описание допустимых переходов между состояниями конечного автомата.

Все параметры объекта модели безопасности Flow являются обязательными.

Пример:

```
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"]
                             }
             }
}
```

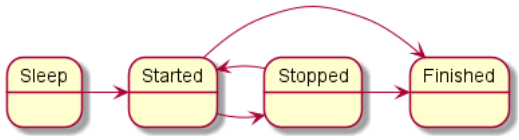


Диаграмма состояний конечного автомата в примере

Объект модели безопасности Flow может быть покрыт аудитом безопасности. При этом можно задать условия выполнения аудита, специфичные для модели безопасности Flow. Для этого в описании конфигурации аудита нужно использовать следующую конструкцию:

`omit : [<"состояние 1">[, ...]]` – аудит не выполняется, если конечный автомат находится в одном из перечисленных состояний.

Необходимость создавать несколько объектов модели безопасности Flow возникает в следующих случаях:

- Если нужно по-разному настроить аудит безопасности для разных объектов модели безопасности Flow (например, для разных объектов можно применять разные профили аудита или разные конфигурации аудита одного профиля).
- Если нужно различать вызовы методов, предоставляемых разными объектами модели безопасности Flow (поскольку в данные аудита включается как имя метода модели безопасности, так и имя объекта, предоставляющего этот метод, можно понять, что был вызван метод конкретного объекта).
- Если нужно использовать конечные автоматы с разными конфигурациями.

Правило init модели безопасности Flow

```
init {sid : <Sid>}
```

Создает конечный автомат и ассоциирует его с ресурсом `sid`. Созданный конечный автомат имеет конфигурацию, заданную в параметрах используемого объекта модели безопасности Flow.

Возвращает результат "разрешено", если создало ассоциацию конечного автомата с ресурсом `sid`.

Возвращает результат "запрещено" в следующих случаях:

- Ресурс `sid` уже ассоциирован с конечным автоматом используемого объекта модели безопасности Flow.
- Значение `sid` вне допустимого диапазона.

Пример:

```
/* Запуск процесса класса Server будет разрешен,
 * если при инициации запуска будет создана
 * ассоциация этого процесса с конечным автоматом.
 * Иначе запуск процесса класса Server будет запрещен. */
execute dst=Server {
    service_flow.init {sid : dst_sid}
}
```

Правило fini модели безопасности Flow

```
fini {sid : <Sid>}
```

Удаляет ассоциацию конечного автомата ресурсом `sid`. Конечный автомат, который более не ассоциирован с ресурсом, уничтожается.

Возвращает результат "разрешено", если удалило ассоциацию конечного автомата с ресурсом `sid`.

Возвращает результат "запрещено" в следующих случаях:

- Ресурс `sid` не ассоциирован с конечным автоматом используемого объекта модели безопасности Flow.
- Значение `sid` вне допустимого диапазона.

Правило enter модели безопасности Flow

```
enter {sid : <Sid>, state : <State>}
```

Переводит конечный автомат, ассоциированный с ресурсом `sid`, в состояние `state`.

Возвращает результат "разрешено", если перевело конечный автомат, ассоциированный с ресурсом `sid`, в состояние `state`.

Возвращает результат "запрещено" в следующих случаях:

- Переход в состояние `state` из текущего состояния не допускается конфигурацией конечного автомата, ассоциированного с ресурсом `sid`.
- Ресурс `sid` не ассоциирован с конечным автоматом используемого объекта модели безопасности Flow.
- Значение `sid` вне допустимого диапазона.

Пример:

```
/* Любому клиенту в решении будет разрешено обращаться
 * к серверу класса Server, если конечный автомат,
```

```
 * ассоциированный с этим сервером, будет переведен в
 * состояние started при инициации обращения. Иначе
 * любому клиенту в решении будет запрещено обращаться
 * к серверу класса Server. */
request dst=Server {
    service_flow.enter {sid : dst_sid, state : "started"}
}
```

Правило allow модели безопасности Flow

```
allow {sid : <Sid>, states : <Set<State>>}
```

Проверяет, что состояние конечного автомата, ассоциированного с ресурсом `sid`, входит в набор состояний `states`.

Возвращает результат "разрешено", если состояние конечного автомата, ассоциированного с ресурсом `sid`, входит в набор состояний `states`.

Возвращает результат "запрещено" в следующих случаях:

- Состояние конечного автомата, ассоциированного с ресурсом `sid`, не входит в набор состояний `states`.
- Ресурс `sid` не ассоциирован с конечным автоматом используемого объекта модели безопасности Flow.
- Значение `sid` вне допустимого диапазона.

Пример:

```
/* Любому клиенту в решении разрешено обращаться к серверу класса
 * Server, если конечный автомат, ассоциированный с этим сервером,
 * находится в состоянии started или stopped. Иначе любому клиенту
 * в решении запрещено обращаться к серверу класса Server. */
request dst=Server {
    service_flow.allow {sid : dst_sid, states : ["started", "stopped"]}
}
```

Выражение query модели безопасности Flow

```
query {sid : <Sid>}
```

Предназначено для использования в качестве выражения, проверяющего выполнение условий в конструкции `choice` (о конструкции `choice` см. "Привязка методов моделей безопасности к событиям безопасности"). Проверяет состояние конечного автомата, ассоциированного с ресурсом `sid`. В зависимости от результатов этой проверки выполняются различные варианты обработки события безопасности.

Выполняется некорректно в следующих случаях:

- Ресурс `sid` не ассоциирован с конечным автоматом используемого объекта модели безопасности Flow.
- Значение `sid` вне допустимого диапазона.

Когда выражение выполняется некорректно, модуль безопасности Kaspersky Security Module возвращает решение "запрещено".

Пример:

```
/* Любому клиенту в решении разрешено обращаться к
 * серверу класса ResourceDriver, если конечный автомат,
 * ассоциированный с этим сервером, находится в состоянии
 * started или stopped. Иначе любому клиенту в решении
 * запрещено обращаться к серверу класса ResourceDriver. */
request dst=ResourceDriver {
    choice (service_flow.query {sid : dst_sid}) {
        "started"   : grant ()
        "stopped"   : grant ()
        _           : deny ()
    }
}
```

Модель безопасности Mic

Модель безопасности Mic позволяет реализовать мандатный контроль целостности. То есть эта модель безопасности дает возможность управлять информационными потоками между процессами, а также между процессами и ядром KasperskyOS, контролируя уровни целостности процессов, ядра и ресурсов, используемых через IPC.

> В терминах модели безопасности Mic процессы и ядро называются субъектами, а ресурсы называются объектами. Однако сведения в этом разделе приведены с отступлением от терминологии модели безопасности Mic. Это отступление заключается в том, что термин "объект" не используется в значении "ресурс".

Информационные потоки между субъектами возникают, когда субъекты взаимодействуют через IPC.

Уровень целостности субъекта/ресурса – это степень доверия к субъекту/ресурсу. Степень доверия к субъекту определяется, например, исходя из того, взаимодействует ли субъект с недоверенными внешними программно-аппаратными системами, или имеет ли субъект доказанный уровень качества. (Ядро имеет высокий уровень целостности.) Степень доверия к ресурсу определяется, например, с учетом того, был ли этот ресурс создан доверенным субъектом внутри программно-аппаратной системы под управлением KasperskyOS или получен из недоверенной внешней программно-аппаратной системы.

Модель безопасности Mic характеризуют следующие положения:

- По умолчанию информационные потоки от менее целостных к более целостным субъектам запрещены. Опционально такие информационные потоки могут быть разрешены. При этом нужно гарантировать, что более целостные субъекты не будут компрометированы.
- Потребителю ресурсов запрещено записывать данные в ресурс, если уровень целостности ресурса выше уровня целостности потребителя ресурсов.

- По умолчанию потребителю ресурсов запрещено читать данные из ресурса, если уровень целостности ресурса ниже уровня целостности потребителя ресурсов. Опционально потребителю ресурсов может быть разрешена такая операция. При этом нужно гарантировать, что потребитель ресурсов не будет компрометирован.

Методы модели безопасности Mic позволяют выполнять следующие операции:

- назначать субъектам и ресурсам уровни целостности;
- отменять назначение уровня целостности для ресурсов;
- проверять допустимость информационных потоков на основе сравнения уровней целостности;
- повышать уровни целостности ресурсов.

PSL-файл с описанием модели безопасности Mic находится в KasperskyOS SDK по пути:

`toolchain/include/nk/mic.psl`

В качестве примера использования модели безопасности Mic можно рассмотреть безопасное обновление ПО программно-аппаратной системы под управлением KasperskyOS. В обновлении участвуют четыре процесса:

- `Downloader` – низкоцелостный процесс, который загружает низкоцелостный образ обновления с удаленного сервера в интернете.
- `Verifier` – высокоцелостный процесс, который проверяет цифровую подпись низкоцелостного образа обновления (высокоцелостный процесс, который может читать данные из низкоцелостного ресурса).
- `FileSystem` – высокоцелостный процесс, который управляет файловой системой.
- `Updater` – высокоцелостный процесс, который применяет обновление.

Обновление ПО выполняется по следующему сценарию:

1. `Downloader` загружает образ обновления и сохраняет его в файл, передав содержимое образа в `FileSystem`. Этому файлу назначается низкий уровень целостности.
2. `Verifier` получает образ обновления у `FileSystem`, прочитав низкоцелостный файл, и проверяет его цифровую подпись. Если подпись корректна, `Verifier` обращается к `FileSystem`, чтобы `FileSystem` создал копию файла с образом обновления. Новому файлу назначается высокий уровень целостности.
3. `Updater` получает образ обновления у `FileSystem`, прочитав высокоцелостный файл, и применяет обновление.

В этом примере модель безопасности Mic обеспечивает то, что высокоцелостный процесс `Updater` может читать данные только из высокоцелостного образа обновления. Вследствие этого обновление может быть применено только после проверки цифровой подписи образа обновления.

Объект модели безопасности Mic

Чтобы использовать модель безопасности Mic, нужно создать объект (объекты) этой модели. При этом нужно задать множество уровней целостности субъектов/ресурсов.

Объект модели безопасности Mic имеет следующие параметры:

- `config` – множество уровней целостности или конфигурация множества уровней целостности:
  - `degrees` – множество градаций для формирования множества уровней целостности;
  - `categories` – множество категорий для формирования множества уровней целостности.

Примеры:

```
policy object mic : Mic {
    config = ["LOW", "MEDIUM", "HIGH"]
}

policy object mic_po : Mic {
    config =
      { degrees    : ["low", "high"]
      , categories : ["net", "log"]
      }
}
```

Множество уровней целостности представляет собой частично упорядоченное множество, которое является линейно упорядоченным или содержит несравнимые элементы. Множество {LOW, MEDIUM, HIGH} является линейно упорядоченным, так как все его элементы сравнимы между собой. Несравнимые элементы появляются, когда множество уровней целостности задается через множество градаций и множество категорий. В этом случае множество уровней целостности *L* представляет собой декартово произведение булеана множества категорий *C* на множество градаций *D*:

$$L = 2^C \times D.$$

Параметры `degrees` и `categories` в примере задают следующее множество:

{

{}/low, {}/high,

{net}/low, {net}/high,

{log}/low, {log}/high,

{net,log}/low, {net,log}/high

}

В этом множестве {} означает пустое множество.

Отношение порядка между элементами множества уровней целостности *L* задается следующим образом:

$$l_i = A/B,$$

$$l_j = E/F,$$

$$l_i \prec l_j \Leftrightarrow \begin{cases} A \subseteq E, \\ B \leq F. \end{cases}$$

Согласно этому отношению порядка *j*-й элемент превышает *i*-й элемент, если подмножество категорий *E* включает подмножество категорий *A*, и градация *F* больше градации *A* либо равна ей. Примеры сравнения элементов множества уровней целостности *L*:

- Элемент {net,log}/high превышает элемент {log}/low, так как градация high больше градации low, и подмножество категорий {net,log} включает подмножество категорий {log}.
- Элемент {net,log}/low превышает элемент {log}/low, так как уровни градаций для этих элементов равны между собой, и подмножество категорий {net,log} включает подмножество категорий {log}.
- Элемент {net,log}/high является наибольшим, так как превышает все остальные элементы.
- Элемент {}/low является наименьшим, так как все остальные элементы превышают этот элемент.
- Элементы {net}/low и {log}/high являются несравнимыми, так как градация high больше градации low, но подмножество категорий {log} не включает подмножество категорий {net}.
- Элементы {net,log}/low и {log}/high являются несравнимыми, так как градация high больше градации low, но подмножество категорий {log} не включает подмножество категорий {net,log}.

> Для субъектов и ресурсов с несравнимыми уровнями целостности модель безопасности Mic предусматривает условия, аналогичные тем, которые эта модель безопасности предусматривает для субъектов и ресурсов со сравнимыми уровнями целостности.
> По умолчанию информационные потоки между субъектам с несравнимыми уровнями целостности запрещены, но опционально такие информационные потоки могут быть разрешены. (Нужно гарантировать, что субъекты, принимающие данные, не будут компрометированы.) Потребителю ресурсов запрещено записывать данные в ресурс и читать данные из ресурса, если уровень целостности ресурса несравним с уровнем целостности потребителя ресурсов. Опционально потребителю ресурсов может быть разрешено чтение данных из ресурса. (Нужно гарантировать, что потребитель ресурсов не будет компрометирован.)

Объект модели безопасности Mic может быть покрыт аудитом безопасности. Условия выполнения аудита, специфичные для модели безопасности Mic, отсутствуют.

Необходимость создавать несколько объектов модели безопасности Mic возникает в следующих случаях:

- Если нужно по-разному настроить аудит безопасности для разных объектов модели безопасности Mic (например, для разных объектов можно применять разные профили аудита или разные конфигурации аудита одного профиля).
- Если нужно различать вызовы методов, предоставляемых разными объектами модели безопасности Mic (поскольку в данные аудита включается как имя метода модели безопасности, так и имя объекта, предоставляющего этот метод, можно понять, что был вызван метод конкретного объекта).
- Если нужно использовать несколько вариантов мандатного контроля целостности, например, с разными множествами уровней целостности субъектов/ресурсов.

Правило create модели безопасности Mic

```
create { source    : <Sid>
       , target    : <Sid>
       , container : <Sid | ()>
       , driver    : <Sid>
       , level     : <Level | ... | ()>
       }
```

Назначает ресурсу `target` уровень целостности `level` в следующей ситуации:

- Процесс `source` инициирует создание ресурса `target`.
- Ресурсом `target` управляет субъект `driver`, который является поставщиком ресурсов или ядром KasperskyOS.
- Ресурс `container` является контейнером для ресурса `target` (например, директория является контейнером для файлов и/или других директорий).

Если поле `container` имеет значение `()`, ресурс `target` рассматривается как корневой, то есть не имеющий контейнера.

Чтобы задать уровень целостности `level`, используются значения типа `Level`:

```
type Level = LevelFull | LevelNoCategory

type LevelFull =
  { degree     : Text | ()
  , categories : List<Text> | ()
  }

type LevelNoCategory = Text
```

Правило возвращает результат "разрешено", если назначило ресурсу `target` уровень целостности `level`.

Правило возвращает результат "запрещено" в следующих случаях:

- Значение `level` превышает уровень целостности процесса `source`, субъекта `driver` или ресурса `container`.
- Значение `level` несравнимо с уровнем целостности процесса `source`, субъекта `driver` или ресурса `container`.
- Процессу `source`, субъекту `driver` или ресурсу `container` не назначен уровень целостности.
- Значение `source`, `target`, `container` или `driver` вне допустимого диапазона.

Пример:

```
/* Серверу класса updater.Realmserv будет разрешено отвечать на
 * обращения любого клиента в решении, вызывающего метод resolve
 * службы realm.Reader, если ресурсу, создание которого запрашивает
 * клиент, при инициации ответа будет назначен уровень целостности LOW.
 * Иначе серверу класса updater.Realmserv будет запрещено отвечать на
 * обращения любого клиента, вызывающего метод resolve службы realm.Reader. */
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"
                            }
```

```
            }
}
```

Правило delete модели безопасности Mic

```
delete { source    : <Sid>
       , target    : <Sid>
       , container : <Sid | ()>
       , driver    : <Sid>
       }
```

Отменяет назначение уровня целостности для ресурса `target` в следующей ситуации:

- Процесс `source` инициирует удаление ресурса `target`.
- Ресурсом `target` управляет субъект `driver`, который является поставщиком ресурсов или ядром KasperskyOS.
- Ресурс `container` является контейнером для ресурса `target` (например, директория является контейнером для файлов и/или других директорий).

Если поле `container` имеет значение `()`, ресурс `target` рассматривается как корневой, то есть не имеющий контейнера.

Правило возвращает результат "разрешено", если отменило назначение уровня целостности для ресурса `target`.

Правило возвращает результат "запрещено" в следующих случаях:

- Уровень целостности ресурса `target` превышает уровень целостности процесса `source` или субъекта `driver`.
- Уровень целостности ресурса `target` несравним с уровнем целостности процесса `source` или субъекта `driver`.
- Процессу `source`, субъекту `driver`, ресурсу `target` или ресурсу `container` не назначен уровень целостности.
- Значение `source`, `target`, `container` или `driver` вне допустимого диапазона.

Пример:

```
/* Любому клиенту в решении будет разрешено обращаться к серверу класса
 * updater.Realmserv, вызывая метод del службы realm.Reader, если для ресурса,
 * удаление которого запрашивает клиент, будет отменено назначение уровня целостности.
 * Иначе любому клиенту в решении будет запрещено обращаться к серверу класса
 * updater.Realmserv, вызывая метод del службы realm.Reader. */
request dst=updater.Realmserv,
        endpoint=realm.Reader {
            match method=del {
                mic.delete { source : src_sid
                           , target : message.handle.handle
```

```
                        , container : ()
                        , driver : dst_sid
                        }
            }
}
```

Правило execute модели безопасности Mic

```
execute <ExecuteImage | ExecuteLevel>

type ExecuteImage =
  { image  : Sid
  , target : Sid
  , level  : Level | ... | ()
  , levelR : Level | ... | ()
  }

type ExecuteLevel =
  { image  : Sid | ()
  , target : Sid
  , level  : Level | ...
  , levelR : Level | ... | ()
  }
```

Назначает субъекту `target` уровень целостности `level` и задает минимальный уровень целостности субъектов и ресурсов, из которых этот субъект может принимать данные (`levelR`). Код субъекта `target` содержится в исполняемом файле `image`.

Если поле `level` имеет значение `()`, субъекту `target` назначается уровень целостности исполняемого файла `image`. Если поле `image` имеет значение `()`, поле `level` должно иметь значение, отличное от `()`.

Если поле `levelR` имеет значение `()`, уровень целостности `levelR` принимается равным уровню целостности субъекта `target`.

Чтобы задать уровни целостности `level` и `levelR`, используются значения типа `Level`. Определение типа `Level` см. в "Правило create модели безопасности Mic".

Правило возвращает результат "разрешено", если назначило субъекту `target` уровень целостности `level` и задало минимальный уровень целостности субъектов и ресурсов, из которых этот субъект может принимать данные (`levelR`).

Правило возвращает результат "запрещено" в следующих случаях:

- Значение `level` превышает уровень целостности исполняемого файла `image`.
- Значение `level` несравнимо с уровнем целостности исполняемого файла `image`.
- Значение `levelR` превышает значение `level`.
- Значения `level` и `levelR` несравнимы.
- Исполняемому файлу `image` не назначен уровень целостности.

- Значение `image` или `target` вне допустимого диапазона.

Пример:

```
/* Запуск процесса класса updater.Manager будет разрешен,
 * если при инициации запуска этому процессу будет назначен
 * уровень целостности LOW, а также будет задан минимальный
 * уровень целостности процессов и ресурсов, из которых этот
 * процесс может принимать данные (LOW). Иначе запуск процесса
 * класса updater.Manager будет запрещен. */
execute src=Einit, dst=updater.Manager, method=main {
    mic.execute { target : dst_sid
                , image : ()
                , level : "LOW"
                , levelR : "LOW"
                }
}
```

Правило upgrade модели безопасности Mic

```
upgrade { source    : <Sid>
        , target    : <Sid>
        , container : <Sid | ()>
        , driver    : <Sid>
        , level     : <Level | ...>
        }
```

Повышает назначенный ранее уровень целостности ресурса `target` до значения `level` в следующей ситуации:

- Процесс `source` инициирует повышение уровня целостности ресурса `target`.
- Ресурсом `target` управляет субъект `driver`, который является поставщиком ресурсов или ядром KasperskyOS.
- Ресурс `container` является контейнером для ресурса `target` (например, директория является контейнером для файлов и/или других директорий).

Если поле `container` имеет значение `()`, ресурс `target` рассматривается как корневой, то есть не имеющий контейнера.

Чтобы задать уровень целостности `level`, используются значения типа `Level`. Определение типа `Level` см. в "Правило create модели безопасности Mic".

Правило возвращает результат "разрешено", если повысило назначенный ранее уровень целостности ресурса `target` до значения `level`.

Правило возвращает результат "запрещено" в следующих случаях:

- Значение `level` не превышает уровень целостности ресурса `target`.

- Значение `level` превышает уровень целостности процесса `source`, субъекта `driver` или ресурса `container`.
- Уровень целостности ресурса `target` превышает уровень целостности процесса `source`.
- Процессу `source`, субъекту `driver` или ресурсу `container` не назначен уровень целостности.
- Значение `source`, `target`, `container` или `driver` вне допустимого диапазона.

Правило call модели безопасности Mic

```
call {source : <Sid>, target : <Sid>}
```

Проверяет допустимость информационных потоков от субъекта `target` к субъекту `source`.

Возвращает результат "разрешено" в следующих случаях:

- Уровень целостности субъекта `source` не превышает уровень целостности субъекта `target`.
- Уровень целостности субъекта `source` превышает уровень целостности субъекта `target`, но минимальный уровень целостности субъектов и ресурсов, из которых субъект `source` может принимать данные, не превышает уровень целостности субъекта `target`.
- Уровень целостности субъекта `source` несравним с уровнем целостности субъекта `target`, но минимальный уровень целостности субъектов и ресурсов, из которых субъект `source` может принимать данные, не превышает уровень целостности субъекта `target`.

Возвращает результат "запрещено" в следующих случаях:

- Уровень целостности субъекта `source` превышает уровень целостности субъекта `target`, и минимальный уровень целостности субъектов и ресурсов, из которых субъект `source` может принимать данные, превышает уровень целостности субъекта `target`.
- Уровень целостности субъекта `source` превышает уровень целостности субъекта `target`, и минимальный уровень целостности субъектов и ресурсов, из которых субъект `source` может читать данные, несравним с уровнем целостности субъекта `target`.
- Уровень целостности субъекта `source` несравним с уровнем целостности субъекта `target`, и минимальный уровень целостности субъектов и ресурсов, из которых субъект `source` может принимать данные, превышает уровень целостности субъекта `target`.
- Уровень целостности субъекта `source` несравним с уровнем целостности субъекта `target`, и минимальный уровень целостности субъектов и ресурсов, из которых субъект `source` может принимать данные, несравним с уровнем целостности субъекта `target`.
- Субъекту `source` или субъекту `target` не назначен уровень целостности.
- Значение `source` или `target` вне допустимого диапазона.

Пример:

```
/* Любому клиенту в решении разрешено обращаться к
 * любому серверу (ядру), если информационные потоки от
 * сервера (ядра) к клиенту допускаются моделью
 * безопасности Mic. Иначе любому клиенту в решении
 * запрещено обращаться к любому серверу (ядру). */
request {
    mic.call { source : src_sid
             , target : dst_sid
             }
}
```

Правило invoke модели безопасности Mic

```
invoke {source : <Sid>, target : <Sid>}
```

Проверяет допустимость информационных потоков от субъекта `source` к субъекту `target`.

Возвращает результат "разрешено", если уровень целостности субъекта `target` не превышает уровень целостности субъекта `source`.

Возвращает результат "запрещено" в следующих случаях:

- Уровень целостности субъекта `target` превышает уровень целостности субъекта `source`.
- Уровень целостности субъекта `target` несравним с уровнем целостности субъекта `source`.
- Субъекту `source` или субъекту `target` не назначен уровень целостности.
- Значение `source` или `target` вне допустимого диапазона.

Правило read модели безопасности Mic

```
read {source : <Sid>, target : <Sid>}
```

Проверяет допустимость чтения данных из ресурса `target` потребителем ресурсов `source`.

Возвращает результат "разрешено" в следующих случаях:

- Уровень целостности потребителя ресурсов `source` не превышает уровень целостности ресурса `target`.
- Уровень целостности потребителя ресурсов `source` превышает уровень целостности ресурса `target`, но минимальный уровень целостности субъектов и ресурсов, из которых потребитель ресурсов `source` может принимать данные, не превышает уровень целостности ресурса `target`.
- Уровень целостности потребителя ресурсов `source` несравним с уровнем целостности ресурса `target`, но минимальный уровень целостности субъектов и ресурсов, из которых потребитель ресурсов `source`

может принимать данные, не превышает уровень целостности ресурса `target`.

Возвращает результат "запрещено" в следующих случаях:

- Уровень целостности потребителя ресурсов `source` превышает уровень целостности ресурса `target`, и минимальный уровень целостности субъектов и ресурсов, из которых потребитель ресурсов `source` может принимать данные, превышает уровень целостности ресурса `target`.
- Уровень целостности потребителя ресурсов `source` превышает уровень целостности ресурса `target`, и минимальный уровень целостности субъектов и ресурсов, из которых потребитель ресурсов `source` может принимать данные, несравним с уровнем целостности ресурса `target`.
- Уровень целостности потребителя ресурсов `source` несравним с уровнем целостности ресурса `target`, и минимальный уровень целостности субъектов и ресурсов, из которых потребитель ресурсов `source` может принимать данные, превышает уровень целостности ресурса `target`.
- Уровень целостности потребителя ресурсов `source` несравним с уровнем целостности ресурса `target`, и минимальный уровень целостности субъектов и ресурсов, из которых потребитель ресурсов `source` может принимать данные, несравним с уровнем целостности ресурса `target`.
- Потребителю ресурсов `source` или ресурсу `target` не назначен уровень целостности.
- Значение `source` или `target` вне допустимого диапазона.

Пример:

```
/* Любому клиенту в решении разрешено обращаться к серверу
 * класса updater.Realmserv, вызывая метод read службы
 * realm.Reader, если модель безопасности Mic допускает
 * чтение данных этим клиентом из ресурса, который требуется
 * этому клиенту. Иначе любому клиенту в решении запрещено
 * обращаться к серверу класса updater.Realmserv, вызывая
 * метод read службы realm.Reader. */
request dst=updater.Realmserv,
        endpoint=realm.Reader {
            match method=read {
                mic.read { source : src_sid
                         , target : message.handle.handle
                         }
               }
}
```

Правило write модели безопасности Mic

```
write {source : <Sid>, target : <Sid>}
```

Проверяет допустимость записи данных в ресурс `target` потребителем ресурсов `source`.

Возвращает результат "разрешено", если уровень целостности ресурса `target` не превышает уровень целостности потребителя ресурсов `source`.

Возвращает результат "запрещено" в следующих случаях:

- Уровень целостности ресурса `target` превышает уровень целостности потребителя ресурсов `source`.
- Уровень целостности ресурса `target` несравним с уровнем целостности потребителя ресурсов `source`.
- Потребителю ресурсов `source` или ресурсу `target` не назначен уровень целостности.
- Значение `source` или `target` вне допустимого диапазона.

Выражение query_level модели безопасности Mic

```
query_level {source : <Sid>}
```

Предназначено для использования в качестве выражения, проверяющего выполнение условий в конструкции `choice` (о конструкции `choice` см. "Привязка методов моделей безопасности к событиям безопасности"). Проверяет уровень целостности субъекта или ресурса `source`. В зависимости от результатов этой проверки выполняются различные варианты обработки события безопасности.

Выполняется некорректно в следующих случаях:

- Субъекту или ресурсу `source` не назначен уровень целостности.
- Значение `source` вне допустимого диапазона.

Когда выражение выполняется некорректно, модуль безопасности Kaspersky Security Module возвращает решение "запрещено".

Методы служб ядра KasperskyOS

С точки зрения модуля безопасности Kaspersky Security Module ядро KasperskyOS является контейнером компонентов, предоставляющих службы. Список компонентов ядра содержится в файле `Core.edl`, расположенном в директории `sysroot-*-kos/include/kl/core` из состава KasperskyOS SDK. Также в этой директории находятся CDL-, IDL-файлы формальной спецификации ядра.

Методы служб ядра можно разделить на безопасные и потенциально опасные. Потенциально опасные методы могут быть использованы злоумышленником в компрометированном компоненте решения, чтобы, например, вызвать отказ, организовать скрытую передачу данных, захватить управление устройством ввода-вывода. Безопасные методы не могут быть использованы таким образом.

Доступ к методам служб ядра должен быть максимально ограничен политикой безопасности решения (принцип "Least Privilege"). Для этого нужно выполнить следующие требования:

1. Разрешить доступ к безопасному методу только тем компонентам решения, которым этот метод нужен.
2. Разрешить доступ к потенциально опасному методу только тем доверенным компонентам решения, которым этот метод нужен.
3. Разрешить доступ к потенциально опасному методу только тем недоверенным компонентам решения, которым этот метод нужен, если только проверяемые условия доступа ограничивают возможности злонамеренного использования метода, или последствия злонамеренного использования метода допустимы с точки зрения безопасности.

Например, недоверенному компоненту можно разрешить использовать ограниченный набор портов ввода-вывода, чтобы этот компонент не мог захватить управление устройствами ввода-вывода. Также, к примеру, скрытая передача данных между недоверенными компонентами может быть допустимой с точки зрения безопасности.

Служба виртуальной памяти

Служба предназначена для управления виртуальной памятью.

Сведения о методах службы приведены в таблице ниже.

Методы службы vmm.VMM (интерфейс kl.core.VMM)

Метод	Назначение и параметры метода	Потенциальная опасность метода
`Allocate`	Назначение Выделяет (резервирует и опционально фиксирует) регион виртуальной памяти. Параметры • [in] `addr` – желаемый базовый адрес региона виртуальной памяти или `0`, чтобы базовый адрес был выбран автоматически. • [in] `size` – размер региона виртуальной памяти в байтах. • [in] `flags` – флаги, задающие параметры региона виртуальной памяти. • [out] `va` – базовый адрес выделенного региона виртуальной памяти. • [out] `rc` – код возврата.	Позволяет выполнить следующие действия: • Исчерпать память ядра, создавая в ней множество объектов. • Исчерпать оперативную память.
`Commit`	Назначение Фиксирует зарезервированный методом `Allocate` регион виртуальной памяти. Параметры • [in] `va` – базовый адрес региона виртуальной памяти. • [in] `size` – размер региона виртуальной памяти в байтах.	Позволяет исчерпать оперативную память.

	• [in] `flags` – фиктивный параметр. • [out] `rc` – код возврата.	
`Decommit`	Назначение Отменяет фиксацию региона виртуальной памяти. Параметры • [in] `va` – базовый адрес региона виртуальной памяти. • [in] `size` – размер региона виртуальной памяти в байтах. • [out] `rc` – код возврата.	Нет.
`Protect`	Назначение Изменяет права доступа к региону виртуальной памяти. Параметры • [in] `va` – базовый адрес региона виртуальной памяти. • [in] `size` – размер региона виртуальной памяти в байтах. • [in] `flags` – флаги, задающие права доступа к региону виртуальной памяти. • [out] `rc` – код возврата.	Нет.
`Free`	Назначение Освобождает регион виртуальной памяти. Параметры • [in] `va` – базовый адрес региона виртуальной памяти. • [in] `size` – размер региона виртуальной памяти в байтах. • [out] `rc` – код возврата.	Нет.
`Query`	Назначение	Нет.

	Позволяет получить сведения о странице виртуальной памяти. Параметры • [in] `va` – адрес, входящий в страницу виртуальной памяти. • [out] `info` – последовательность, содержащая сведения о странице виртуальной памяти. • [out] `rc` – код возврата.	
`MdlCreate`	Назначение Создает буфер MDL. Параметры • [in] `size` – размер буфера MDL в байтах. • [in] `prot` – флаги, задающие права доступа к буферу MDL. • [out] `handle` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует буфер MDL. • [out] `rc` – код возврата.	Позволяет выполнить следующие действия: • Исчерпать память ядра, создавая в ней множество объектов. • Исчерпать оперативную память.
`MdlCreateFromVm`	Назначение Создает буфер MDL из физической памяти, отображенной на заданный регион виртуальной памяти, и отображает созданный буфер MDL на этот регион. Параметры • [in] `va` – базовый адрес региона виртуальной памяти. • [in] `size` – размер региона виртуальной памяти в байтах. • [in] `flags` – флаги, задающие права доступа к буферу MDL.	Позволяет выполнить следующие действия: • Исчерпать память ядра, создавая в ней множество объектов. • Исчерпать оперативную память.

	• [out] `handle` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует буфер MDL. • [out] `rc` – код возврата.	
`MdlGetSize`	Назначение Позволяет получить размер буфера MDL. Параметры • [in] `handle` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует буфер MDL. • [out] `size` – размер буфера MDL в байтах. • [out] `rc` – код возврата.	Нет.
`MdlMap`	Назначение Резервирует регион виртуальной памяти и отображает на него буфер MDL. Параметры • [in] `handle` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует буфер MDL. • [in] `offset` – смещение в буфере MDL, с которого нужно начать отображение, в байтах. • [in] `length` – размер части буфера MDL, которую нужно отобразить, в байтах. • [in] `hint` – желаемый базовый адрес региона виртуальной памяти или `0`, чтобы базовый адрес был выбран автоматически.	Позволяет выполнить следующие действия: • Создать разделяемую память для межпроцессного взаимодействия, скрытого от модуля безопасности, если дескрипторами одного буфера MDL владеют несколько процессов (маски прав дескрипторов должны разрешать отображение буфера MDL). • Исчерпать память ядра, создавая в ней множество объектов.

	• [in] `prot` – флаги, задающие параметры региона виртуальной памяти. • [out] `address` – базовый адрес региона виртуальной памяти. • [out] `rc` – код возврата.	
`MdlClone`	Назначение Создает буфер MDL на основе существующего. Буфер MDL создается из тех же регионов физической памяти, что и оригинальный. Параметры • [in] `originHandle` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует оригинальный буфер MDL. • [in] `offset` – смещение в оригинальном буфере MDL, с которого нужно начать дублирование, в байтах. • [in] `length` – размер части оригинального буфера MDL, которую нужно дублировать, в байтах. • [out] `cloneHandle` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует созданный буфер MDL. • [out] `rc` – код возврата.	Позволяет исчерпать память ядра, создавая в ней множество объектов.

Служба ввода-вывода

Служба предназначена для работы с портами ввода-вывода, MMIO, DMA, прерываниями.

Сведения о методах службы приведены в таблице ниже.

Метод	Назначение и параметры метода	Потенциальная опасность метода
`RegisterPort`	Назначение Регистрирует последовательность портов ввода-вывода. Параметры • [in] `base` – адрес первого порта ввода-вывода в последовательности. • [in] `size` – число портов ввода-вывода в последовательности. • [out] `resource` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует последовательность портов ввода-вывода. • [out] `rc` – код возврата.	Позволяет выполнить следующие действия: • Захватить порты ввода-вывода (рекомендуется контролировать адрес первого порта ввода-вывода и число портов ввода-вывода в последовательности). • Исчерпать память ядра, создавая в ней множество объектов.
`RegisterMmio`	Назначение Регистрирует регион памяти MMIO. Параметры • [in] `base` – базовый адрес региона памяти MMIO. • [in] `size` – размер региона памяти MMIO в байтах. • [out] `resource` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует регион памяти MMIO. • [out] `rc` – код возврата.	Позволяет исчерпать память ядра, создавая в ней множество объектов.
`RegisterDma`	Назначение Создает буфер DMA. Параметры • [in] `size` – размер буфера DMA в байтах. • [in] `flags` – флаги, задающие параметры буфера DMA.	Позволяет выполнить следующие действия: • Исчерпать память ядра, создавая в ней множество объектов. • Исчерпать оперативную память.

	• [in] `order` – параметр, задающий минимальное число страниц памяти (*2^order*) в блоке. • [out] `resource` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует буфер DMA. • [out] `rc` – код возврата.	
`RegisterIrq`	Назначение Регистрирует прерывание. Параметры • [in] `irq` – номер прерывания. • [out] `resource` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует прерывание. • [out] `rc` – код возврата.	Позволяет исчерпать память ядра, создавая в ней множество объектов.
`MapMem`	Назначение Резервирует регион виртуальной памяти и отображает на него регион памяти MMIO. Параметры • [in] `resource` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует регион памяти MMIO. • [in] `prot` – флаги, задающие права доступа к региону виртуальной памяти. • [in] `attr` – флаги, задающие параметры региона виртуальной памяти (например, использование кеширования). • [out] `address` – базовый адрес региона виртуальной памяти. • [out] `mapping` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор используется для освобождения региона виртуальной памяти.	Позволяет выполнить следующие действия: • Захватить управление устройством при отображении региона памяти MMIO на регион виртуальной памяти (рекомендуется контролировать базовый адрес и размер региона памяти MMIO при вызове метода `RegisterMmio`). • Создать разделяемую память для межпроцессного взаимодействия, скрытого от модуля безопасности, если дескрипторами одного региона памяти MMIO владеют несколько процессов (маски прав дескрипторов должны разрешать отображение региона памяти MMIO). • Исчерпать память ядра, создавая в ней

	• [out] `rc` – код возврата.	множество объектов.
`PermitPort`	Назначение Открывает доступ к портам ввода-вывода. Параметры • [in] `resource` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует последовательность портов ввода-вывода. • [out] `access` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор используется для закрытия доступа к портам ввода-вывода. • [out] `rc` – код возврата.	Позволяет выполнить следующие действия: • Захватить управление устройством (рекомендуется контролировать адрес первого порта ввода-вывода и число портов ввода-вывода в последовательности при вызове метода `RegisterPort`). • Исчерпать память ядра, создавая в ней множество объектов.
`AttachIrq`	Назначение Привязывает вызывающий поток исполнения к прерыванию. Параметры • [in] `resource` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует прерывание. • [in] `flags` – флаги, задающие параметры прерывания. • [out] `delivery` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор является клиентским IPC-дескриптором, который используется обработчиком прерывания. • [out] `rc` – код возврата.	Позволяет выполнить следующие действия: • Забрать процессорное время у остальных потоков исполнения, в том числе из других процессов (поток исполнения, выполнивший привязку к прерыванию, становится потоком реального времени). • Сделать невозможным завершение процесса из другого процесса (процесс, поток которого выполнил привязку к прерыванию, невозможно завершить из другого процесса). • Остановить операционную систему (при возникновении необработанного исключения в потоке исполнения, обрабатывающем прерывание, останавливается операционная система). • Заблокировать, замедлить или некорректно выполнить

		обработку прерывания (рекомендуется контролировать номер прерывания при вызове метода `RegisterIrq`). • Исчерпать память ядра, создавая в ней множество объектов.
`DetachIrq`	Назначение Отправляет потоку исполнения запрос, в результате выполнения которого поток должен выполнить отвязывание от прерывания. Параметры • [in] `resource` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует прерывание. • [out] `rc` – код возврата.	Позволяет прекратить обработку прерывания в другом процессе.
`EnableIrq`	Назначение Разрешает (демаскирует) прерывание. Параметры • [in] `resource` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует прерывание. • [out] `rc` – код возврата.	Позволяет разрешить прерывание на уровне системы.
`DisableIrq`	Назначение Запрещает (маскирует) прерывание. Параметры • [in] `resource` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует прерывание. • [out] `rc` – код возврата.	Позволяет запретить прерывание на уровне системы.
`ModifyDma`	Назначение Изменяет параметры кеширования буфера DMA. Параметры	Нет.

	• [in] `resource` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует буфер DMA. • [in] `flags` – флаги, задающие параметры кеширования буфера DMA. • [out] `rc` – код возврата.	
`MapDma`	Назначение Резервирует регион виртуальной памяти и отображает на него буфер DMA. Параметры • [in] `resource` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует буфер DMA. • [in] `offset` – смещение в буфере DMA, с которого нужно начать отображение, в байтах. • [in] `length` – размер части буфера DMA, которую нужно отобразить, в байтах. • [in] `hint` – желаемый базовый адрес региона виртуальной памяти или `0`, чтобы базовый адрес был выбран автоматически. • [in] `prot` – флаги, задающие права доступа к региону виртуальной памяти. • [out] `address` – базовый адрес региона виртуальной памяти. • [out] `mapping` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор используется для освобождения региона виртуальной памяти. • [out] `rc` – код возврата.	Позволяет выполнить следующие действия: • Создать разделяемую память для межпроцессного взаимодействия, скрытого от модуля безопасности, если дескрипторами одного буфера DMA владеют несколько процессов (маски прав дескрипторов должны разрешать отображение буфера DMA). • Исчерпать память ядра, создавая в ней множество объектов.
`DmaGetInfo`	Назначение Позволяет получить сведения о буфере DMA. Параметры • [in] `resource` – значение, двоичное представление которого состоит из нескольких	Нет.

	полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует буфер DMA. • [out] `flags` – флаги, отражающие параметры DMA. • [out] `order` – параметр, отражающий минимальное число страниц памяти (2^order) в блоке. • [out] `size` – размер буфера DMA в байтах. • [out] `count` – число блоков. • [out] `frames` – последовательность, содержащая адреса и размеры блоков. • [out] `rc` – код возврата.	
`DmaGetPhysInfo`	Назначение Позволяет получить сведения о физической памяти, на основе которой создан буфер DMA. Параметры • [in] `handle` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует буфер DMA. • [out] `count` – число блоков. • [out] `frames` – последовательность, содержащая адреса и размеры блоков. • [out] `rc` – код возврата.	Нет.
`BeginDma`	Назначение Открывает доступ к буферу DMA для устройства. Параметры • [in] `resource` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует буфер DMA. • [out] `iomapping` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски	Позволяет исчерпать память ядра, создавая в ней множество объектов.

	прав дескриптора. Дескриптор идентифицирует объект ядра, который содержит адреса и размеры блоков, необходимые устройству, чтобы использовать буфер DMA. Адреса памяти, используемые устройством, могут быть физическими или виртуальными в зависимости от того, задействован ли IOMMU. • [out] `rc` – код возврата.	

Служба потоков исполнения

Служба предназначена для управления потоками исполнения.

Сведения о методах службы приведены в таблице ниже.

Методы службы thread.Thread (интерфейс kl.core.Thread)

Метод	Назначение и параметры метода	Потенциальная опасность метода
`Create`	Назначение Создает поток исполнения. Параметры • [out] `thread` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует поток исполнения. • [out] `tid` – идентификатор потока исполнения (TID). • [in] `priority` – приоритет потока исполнения. • [in] `stackSize` – размер стека потока исполнения в байтах или `0`, чтобы был использован размер по умолчанию, заданный при создании процесса. • [in] `routine` – указатель на функцию, вызываемую при запуске потока исполнения. • [in] `context` – указатель на функцию, выполняемую потоком исполнения.	Позволяет выполнить следующие действия: • Создать поток исполнения реального времени, который заберет все процессорное время у остальных потоков исполнения, в том числе из других процессов (рекомендуется контролировать параметры создания потока исполнения). • Создать множество потоков исполнения, в том числе с высоким приоритетом, чтобы сократить процессорное время, доступное потокам других процессов (рекомендуется контролировать приоритет потока исполнения). • Исчерпать оперативную память. • Исчерпать память ядра, создавая в ней множество объектов.

	• [in] `context2` – указатель на параметры, передаваемые функции, заданной через параметр `context`. • [in] `flags` – флаги, задающие параметры создания потока исполнения. • [out] `rc` – код возврата.	
`OpenCurrent`	Назначение Создает дескриптор вызывающего потока исполнения. Параметры • [out] `thread` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует поток исполнения. • [out] `rc` – код возврата.	Нет.
`Suspend`	Назначение Блокирует вызывающий поток исполнения. Параметры • [out] `rc` – код возврата.	Позволяет заблокировать поток исполнения, который захватил созданный в разделяемой памяти объект синхронизации, который ожидается потоком исполнения другого процесса. В результате поток исполнения другого процесса может быть заблокированным сколь угодно долго.
`Resume`	Назначение Возобновляет исполнение заблокированного потока. Параметры • [in] `thread` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует поток исполнения. • [out] `rc` – код возврата.	Нет.

`Terminate`	Назначение Завершает поток исполнения. Параметры • [in] `thread` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует поток исполнения. • [in] `code` – код завершения потока исполнения. • [out] `rc` – код возврата.	Нет.
`Exit`	Назначение Завершает вызывающий поток исполнения. Параметры • [in] `code` – код завершения потока исполнения. • [out] `rc` – код возврата.	Нет.
`Wait`	Назначение Блокирует вызывающий поток исполнения до завершения заданного потока исполнения. Параметры • [in] `thread` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует поток исполнения. • [in] `msec` – время ожидания завершения потока исполнения в миллисекундах. • [out] `code` – код завершения потока исполнения.	Нет.

	• [out] `rc` – код возврата.	
`SetPriority`	Назначение Задает приоритет потока исполнения. Параметры • [in] `thread` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует поток исполнения. • [in] `priority` – приоритет потока исполнения. • [out] `rc` – код возврата.	Позволяет повысить приоритет потока исполнения, чтобы сократить процессорное время, доступное остальным потокам исполнения, в том числе из других процессов. Рекомендуется контролировать приоритет потока исполнения.
`SetTls`	Назначение Задает базовый адрес локальной памяти потока исполнения (TLS) для вызывающего потока исполнения. Параметры • [in] `va` – указатель на локальную память потока исполнения. • [out] `rc` – код возврата.	Нет.
`Sleep`	Назначение Блокирует вызывающий поток исполнения на заданное время. Параметры • [in] `mdelay` – время блокировки потока исполнения в миллисекундах. • [out] `rc` – код возврата.	Нет.
`GetInfo`	Назначение	Нет.

	Позволяет получить сведения о потоке исполнения. Параметры • [in] `thread` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует поток исполнения. • [out] `info` – структура, содержащая базовый адрес стека потока исполнения и его размер в байтах, а также идентификатор потока исполнения (TID). • [out] `rc` – код возврата.	
`DetachIrq`	Назначение Отвязывает вызывающий поток исполнения от прерывания, обрабатываемого в его контексте. Параметры • [out] `rc` – код возврата.	Нет.
`GetAffinity`	Назначение Позволяет получить маску сходства потока исполнения. Параметры • [in] `thread` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует поток исполнения. • [out] `mask` – маска сходства потока исполнения. • [out] `rc` – код возврата.	Нет.

`SetAffinity`	Назначение Задает маску сходства потока исполнения. Параметры • [in] `thread` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует поток исполнения. • [in] `mask` – маска сходства потока исполнения. • [out] `rc` – код возврата.	Нет.
`SetSchedPolicy`	Назначение Задает класс планирования и приоритет потока исполнения. Параметры • [in] `thread` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует поток исполнения. • [in] `policy` – класс планирования потока исполнения. • [in] `priority` – приоритет потока исполнения. • [in] `param` – объединение, содержащее параметры класса планирования потока исполнения. • [out] `rc` – код возврата.	Позволяет выполнить следующие действия: • Сделать поток исполнения потоком исполнения реального времени, который заберет все процессорное время у остальных потоков исполнения, в том числе из других процессов (рекомендуется контролировать класс планирования потока исполнения). • Повысить приоритет потока исполнения, чтобы сократить процессорное время, доступное остальным потокам исполнения, в том числе из других процессов (рекомендуется контролировать приоритет потока исполнения).
`GetSchedPolicy`	Назначение Позволяет получить сведения о классе планирования и приоритете потока исполнения.	Нет.

	Параметры • [in] `thread` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует поток исполнения. • [out] `policy` – класс планирования потока исполнения. • [out] `priority` – приоритет потока исполнения. • [out] `param` – объединение, содержащее параметры класса планирования потока исполнения. • [out] `rc` – код возврата.	

Служба дескрипторов

Служба предназначена для выполнения операций с дескрипторами.

Сведения о методах службы приведены в таблице ниже.

Методы службы handle.Handle (интерфейс kl.core.Handle)

Метод	Назначение и параметры метода	Потенциальная опасность метода
`Copy`	Назначение Копирует дескриптор. В результате копирования вызывающий процесс получает потомка дескриптора. Параметры • [in] `inHandle` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Поле дескриптора содержит оригинальный дескриптор. • [in] `newRightsMask` – маска прав потомка дескриптора. • [in] `copyBadge` – значение, двоичное представление которого состоит из нескольких полей, включая поле	Позволяет исчерпать память ядра, создавая в ней множество объектов.

	дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует объект контекста передачи ресурса. • [out] `outHandle` – значение, двоичное представление которого состоит из нескольких полей, включая поле потомка дескриптора и поле маски прав потомка дескриптора. • [out] `rc` – код возврата.	
`CreateUserObject`	Назначение Создает дескриптор. Параметры • [in] `type` – тип дескриптора. • [in] `rights` – маска прав дескриптора. • [in] `context` – указатель на данные, которые нужно ассоциировать с дескриптором. • [in] `ipcChannel` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор является серверным IPC-дескриптором. • [in] `riid` – идентификатор службы (RIID). • [out] `handle` – значение, двоичное представление которого состоит из нескольких полей, включая поле созданного дескриптора и поле маски прав созданного дескриптора. • [out] `rc` – код возврата.	Позволяет исчерпать память ядра, создавая в ней множество объектов.
`Close`	Назначение Закрывает дескриптор. Параметры • [in] `handle` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. • [out] `rc` – код возврата.	Нет.
`Connect`	Назначение Создает и связывает между собой клиентский, серверный и слушающий IPC-дескрипторы. Параметры	Позволяет исчерпать память ядра, создавая в ней множество объектов.

	• [in] `server` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует серверный процесс. • [in] `srListener` – слушающий дескриптор из пространства дескрипторов серверного процесса или значение `0xFFFFFFFF`, чтобы создать его. • [in] `createSrEndpoint` – значение, которое задает, нужно ли создавать серверный IPC-дескриптор в пространстве дескрипторов серверного процесса (`0` – нет, другое число – да). • [in] `client` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует клиентский процесс. • [out] `outSrListener` – слушающий дескриптор из пространства дескрипторов серверного процесса. • [out] `outSrEndpoint` – серверный IPC-дескриптор из пространства дескрипторов серверного процесса. • [out] `outClEndpoint` – клиентский IPC-дескриптор из пространства дескрипторов клиентского процесса. • [out] `rc` – код возврата.	
`SecurityConnect`	Назначение Создает клиентский IPC-дескриптор для обращения к модулю безопасности Kaspersky Security Module через интерфейс безопасности. Параметры • [out] `client` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. • [out] `rc` – код возврата.	Позволяет исчерпать множество возможных значений дескрипторов процесса ядра.
`GetSidByHandle`	Назначение Позволяет получить идентификатор безопасности (SID) по дескриптору. Параметры • [in] `handle` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. • [out] `sid` – идентификатор безопасности (SID).	Нет.

	• [out] `rc` – код возврата.	
`Revoke`	Назначение Закрывает дескриптор и отзывает его потомков. Параметры • [in] `handle` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. • [out] `rc` – код возврата.	Нет.
`RevokeSubtree`	Назначение Отзывает дескрипторы, которые образуют поддерево наследования заданного дескриптора. Параметры • [in] `handle` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескрипторы, образующие поддерево наследования этого дескриптора, отзываются. • [in] `badge` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует объект контекста передачи ресурса, который определяет поддерево наследования дескрипторов для отзыва. Корневым узлом этого поддерева является дескриптор, который порожден передачей или копированием дескриптора, заданного через параметр `handle`, в ассоциации с объектом контекста передачи ресурса. • [out] `rc` – код возврата.	Нет.
`CreateBadge`	Назначение Создает объект контекста передачи ресурса и настраивает механизм уведомлений для контроля жизненного цикла этого объекта. Параметры • [in] `notify` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует приемник уведомлений. • [in] `notifyContext` – идентификатор записи вида "ресурс – маска событий" в приемнике уведомлений.	Позволяет исчерпать память ядра, создавая в ней множество объектов.

	• [in] `badgeContext` – указатель на данные, которые нужно ассоциировать с передачей дескриптора. • [out] `badge` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует объект контекста передачи ресурса. • [out] `rc` – код возврата.	

Служба процессов

Служба предназначена для управления процессами.

Сведения о методах службы приведены в таблице ниже.

Методы службы task.Task (интерфейс kl.core.Task)

Метод	Назначение и параметры метода	Потенциальная опасность метода
`Create`	Назначение Создает процесс. Параметры • [in] `name` – имя процесса. • [in] `eiid` – имя класса процесса. • [in] `path` – имя исполняемого файла в ROMFS. • [in] `stackSize` – размер стека потока исполнения, используемый по умолчанию при создании потоков процесса, в байтах. • [in] `priority` – приоритет начального потока исполнения. • [in] `flags` – флаги, задающие параметры создания процесса. • [out] `child` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора.	Позволяет выполнить следующие действия: • Создать процесс, который будет привилегированным с точки зрения политики безопасности решения (указав имя класса процессов с привилегиями). • Зарезервировать имя процесса, чтобы другой процесс с таким именем нельзя было создать. • Создать процесс, при возникновении необработанного исключения в котором операционная система останавливается. • Загрузить в память процесса код из исполняемого файла для последующего исполнения. • Исчерпать оперативную память, создавая множество процессов. • Исчерпать память ядра, создавая в ней множество объектов.

	Дескриптор идентифицирует созданный процесс. • [out] `rc` – код возврата.	
`LoadSeg`	Назначение Загружает сегмент ELF-образа в память процесса из буфера MDL. Параметры • [in] `task` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует процесс. • [in] `mdl` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует буфер MDL, содержащий сегмент ELF-образа. • [in] `segAttr` – структура, содержащая параметры загрузки сегмента ELF-образа. • [out] `rc` – код возврата. • [out] `retaddr` – базовый адрес региона виртуальной памяти процесса, куда загружен сегмент ELF-образа.	Позволяет загрузить в память процесса код для последующего исполнения.
`VmReserve`	Назначение Резервирует регион виртуальной памяти в процессе, который был создан "пустым". Параметры • [in] `task` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора.	Позволяет выполнить следующие действия: • Исчерпать память ядра, создавая в ней множество объектов. • Зарезервировать регионы виртуальной памяти в другом процессе, который был создан "пустым" и еще не запущен, при наличии его дескриптора. (Маска прав дескриптора должна

	Дескриптор идентифицирует процесс. • [in] `addr` – желаемый базовый адрес региона виртуальной памяти или `0`, чтобы этот адрес был выбран автоматически. • [in] `size` – размер региона виртуальной памяти в байтах. • [in] `flags` – флаги, задающие параметры региона виртуальной памяти. • [out] `outAddr` – базовый адрес зарезервированного региона виртуальной памяти. • [out] `rc` – код возврата.	разрешать резервирование виртуальной памяти.)
`VmFree`	Назначение Освобождает регион виртуальной памяти, зарезервированный вызовом метода `VmReserve` в процессе, который был создан "пустым". Параметры • [in] `task` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует процесс. • [in] `addr` – базовый адрес региона виртуальной памяти. • [in] `size` – размер региона виртуальной памяти в байтах. • [out] `rc` – код возврата.	Позволяет освободить регионы виртуальной памяти в другом процессе, который был создан "пустым" и еще не запущен, при наличии его дескриптора. (Маска прав дескриптора должна разрешать освобождение виртуальной памяти.)
`SetEntry`	Назначение Задает точку входа в программу и смещение загрузки ELF-образа. Параметры	Создает условия для запуска кода, загруженного в память процесса.

	• [in] `task` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует процесс. • [in] `state` – структура, содержащая адрес точки входа в программу и смещение загрузки ELF-образа в байтах. • [out] `rc` – код возврата.	
`LoadElfSyms`	Назначение Загружает таблицу символов `.symtab` и таблицу строк `.strtab` из буферов MDL в память процесса, который был создан "пустым". Параметры • [in] `task` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует процесс. • [in] `symMdl` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует буфер MDL, содержащий таблицу символов `.symtab`. • [in] `symSegAttr` – структура, содержащая параметры загрузки таблицы символов `.symtab`. • [in] `symSize` – размер таблицы символов `.symtab` в байтах. • [in] `strMdl` – значение, двоичное представление	Нет.

	которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует буфер MDL, содержащий таблицу строк `.strtab`. • [in] `strSegAttr` – структура, содержащая параметры загрузки таблицы строк `.strtab`. • [in] `strSize` – размер таблицы строк `.strtab` в байтах. • [out] `rc` – код возврата.	
`LoadElfHdr`	Назначение Записывает заголовок ELF-образа в PCB процесса, который был создан "пустым". Параметры • [in] `task` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует процесс. • [in] `hdrData` – последовательность, содержащая заголовок ELF-образа. • [out] `rc` – код возврата.	Нет.
`SetEnv`	Назначение Записывает данные в SCP дочернего процесса. Параметры • [in] `task` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора.	Позволяет исчерпать память ядра, создавая в ней множество объектов.

	Дескриптор идентифицирует дочерний процесс. • [in] `env` – последовательность, содержащая данные для записи в SCP. • [out] `rc` – код возврата.	
`FreeSelfEnv`	Назначение Удаляет SCP вызывающего процесса. Параметры • [out] `rc` – код возврата.	Нет.
`Resume`	Назначение Запускает процесс. Параметры • [in] `task` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует процесс. • [out] `rc` – код возврата.	Позволяет выполнить следующие действия: • Запустить на исполнение код, загруженный в память процесса. • Запустить множество ранее созданных процессов, чтобы сократить вычислительные ресурсы, доступные другим процессам (рекомендуется контролировать приоритет начального потока исполнения при вызове метода `Create`).
`Exit`	Назначение Завершает вызывающий процесс. Параметры • [in] `status` – код завершения процесса. • [out] `rc` – код возврата.	Нет.
`Terminate`	Назначение Завершает процесс. Параметры	Позволяет завершить другой процесс при наличии его дескриптора. (Маска прав дескриптора должна разрешать завершение процесса.)

	• [in] `task` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует процесс. • [out] `rc` – код возврата.	
`GetExitInfo`	Назначение Позволяет получить сведения о завершившемся процессе. Параметры • [in] `task` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует завершившейся процесс. • [out] `status` – значение, отражающее причину завершения процесса. • [out] `info` – объединение, содержащее сведения о завершившемся процессе. • [out] `rc` – код возврата.	Нет.
`GetThreadContext`	Назначение Позволяет получить контекст потока исполнения, входящего в процесс, который находится в "замороженном" состоянии. Параметры • [in] `task` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует процесс, который находится в "замороженном" состоянии.	Позволяет нарушить изоляцию процесса, который находится в "замороженном" состоянии. Например, контекст потока исполнения может содержать значения переменных.

	• [in] `index` – индекс потока исполнения. Используется для перечисления потоков исполнения. Нумерация начинается с нуля. Нулевой индекс имеет поток исполнения, в котором возникло необработанное исключение. • [out] `context` – структура, содержащая идентификатор (TID) и контекст потока исполнения. • [out] `rc` – код возврата.	
`GetNextVmRegion`	Назначение Позволяет получить сведения о регионе виртуальной памяти, принадлежащем процессу, который находится "замороженном" состоянии. Параметры • [in] `task` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует процесс, который находится в "замороженном" состоянии. • [in] `after` – адрес, после которого размещен регион виртуальной памяти. • [out] `next` – базовый адрес региона виртуальной памяти. • [out] `size` – размер региона виртуальной памяти в байтах. • [out] `flags` – флаги, отражающие параметры региона виртуальной памяти. • [out] `handle` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора.	Позволяет нарушить изоляцию процесса, который находится в "замороженном" состоянии. Изоляция нарушается, так как открывается доступ к региону памяти процесса.

	Дескриптор идентифицирует буфер MDL, отображенный на регион виртуальной памяти. • [out] `rc` – код возврата.	
`TerminateAfterFreezing`	Назначение Завершает процесс, который находится в "замороженном" состоянии. Параметры • [in] `task` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует процесс, который находится в "замороженном" состоянии. • [out] `rc` – код возврата.	Позволяет завершить процесс, который находится в "замороженном" состоянии. Это не дает получить сведения об этом процессе для диагностики.
`GetName`	Назначение Позволяет получить имя вызывающего процесса. Параметры • [out] `name` – имя процесса. • [out] `rc` – код возврата.	Нет.
`GetPath`	Назначение Позволяет получить имя исполняемого файла (в ROMFS), из которого создан вызывающий процесс. Параметры • [out] `path` – имя исполняемого файла. • [out] `rc` – код возврата.	Нет.
`GetInitialThreadPriority`	Назначение	Нет.

	Позволяет получить приоритет начального потока процесса. Параметры • [in] `task` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует процесс. • [out] `priority` – приоритет начального потока исполнения. • [out] `rc` – код возврата.	
`SetInitialThreadPriority`	Назначение Задает приоритет начального потока процесса. Параметры • [in] `task` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует процесс. • [in] `priority` – приоритет начального потока исполнения. • [out] `rc` – код возврата.	Позволяет повысить приоритет начального потока процесса, чтобы сократить процессорное время, доступное остальным потокам исполнения, в том числе из других процессов. Рекомендуется контролировать приоритет начального потока исполнения.
`GetTasksList`	Назначение Позволяет получить сведения о существующих процессах. Параметры • [out] `notice` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует	Позволяет исчерпать память ядра, создавая в ней множество объектов.

	приемник уведомлений, который настроен на получение уведомлений о завершении процессов. • [out] `strings` – последовательность, содержащая параметры процессов. • [out] `pids` – последовательность, содержащая идентификаторы процессов (PID каждого процесса). • [out] `rc` – код возврата.	
`SetInitialThreadSchedPolicy`	Назначение Задает класс планирования и приоритет начального потока процесса. Параметры • [in] `task` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует процесс. • [in] `policy` – класс планирования начального потока процесса. • [in] `priority` – приоритет начального потока процесса. • [in] `params` – объединение, содержащее параметры класса планирования начального потока процесса. • [out] `rc` – код возврата.	Позволяет выполнить следующие действия: • Сделать начальный поток процесса потоком исполнения реального времени, который заберет все процессорное время у остальных потоков исполнения, в том числе из других процессов (рекомендуется контролировать класс планирования начального потока процесса). • Повысить приоритет начального потока процесса, чтобы сократить процессорное время, доступное остальным потокам исполнения, в том числе из других процессов (рекомендуется контролировать приоритет начального потока процесса).
`ReseedAslr`	Назначение Задает начальное значение генератора случайных чисел для поддержки ASLR. Параметры	Нет.

	• [in] `task` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует процесс. • [in] `seed` – последовательность, содержащая начальное значение генератора случайных чисел. • [out] `rc` – код возврата.	
`GetElfSyms`	Назначение Позволяет получить адрес и размер таблицы символов `.symtab` и таблицы строк `.strtab` для вызывающего процесса. Параметры • [out] `relocBase` – смещение загрузки ELF-образа в байтах. • [out] `syms` – адрес таблицы символов `.symtab`. • [out] `symsCnt` – размер таблицы символов `.symtab` в байтах. • [out] `strs` – адрес таблицы строк `.strtab`. • [out] `strsSize` – размер таблицы строк `.strtab` в байтах. • [out] `rc` – код возврата.	Нет.
`TransferHandle`	Назначение Передает дескриптор процессу, который еще не запущен. Параметры • [in] `task` – значение, двоичное представление которого состоит из	Позволяет исчерпать память ядра, создавая в ней множество объектов.

	нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует процесс. • [in] `srcHandle` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Поле дескриптора содержит передаваемый дескриптор. • [in] `srcBadge` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует объект контекста передачи ресурса. • [in] `dstRights` – маска прав потомка передаваемого дескриптора. • [out] `dstHandle` – значение потомка переданного дескриптора (из пространства дескрипторов процесса, который получил дескриптор). • [out] `rc` – код возврата.	
`GetPid`	Назначение Позволяет получить идентификатор процесса (PID). Параметры • [in] `task` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует процесс. • [out] `pid` – идентификатор процесса.	Нет.

	• [out] `rc` – код возврата.	

Служба синхронизации

Служба предназначена для работы с фьютексами.

Сведения о методах службы приведены в таблице ниже.

Методы службы sync.Sync (интерфейс kl.core.Sync)

Метод	Назначение и параметры метода	Потенциальная опасность метода
`Wait`	Назначение Блокирует исполнение вызывающего потока, если значение фьютекса равно ожидаемому. Параметры • [in] `ptr` – указатель на фьютекс. • [in] `val` – ожидаемое значение фьютекса. • [in] `delay` – максимальное время блокировки в миллисекундах. • [out] `outDelay` – фактическое время блокировки в миллисекундах. • [out] `rc` – код возврата.	Нет.
`Wake`	Назначение Возобновляет исполнение потоков, заблокированных вызовом метода `Wait` с заданным фьютексом. Параметры • [in] `ptr` – указатель на фьютекс. • [in] `nThreads` – максимальное число потоков, исполнение которых может быть возобновлено. • [out] `wokenCnt` – фактическое число потоков, исполнение которых возобновлено. • [out] `rc` – код возврата.	Нет.

Службы файловой системы

Службы предназначены для работы с файловой системой ROMFS, используемой ядром KasperskyOS.

Сведения о методах служб приведены в таблицах ниже.

Методы службы fs.FS (интерфейс kl.core.FS)

Метод	Назначение и параметры метода	Потенциальная опасность метода
`Open`	Назначение Открывает файл. Параметры • [in] `name` – имя файла. • [out] `handle` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует открытый файл. • [out] `rc` – код возврата.	Позволяет исчерпать память ядра, создавая в ней множество объектов.
`Close`	Назначение Закрывает файл. Параметры • [in] `handle` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует открытый файл. • [out] `rc` – код возврата.	Нет.
`Read`	Назначение Читает данные из файла. Параметры • [in] `handle` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует открытый файл. • [in] `sectorNumber` – номер блока данных. Нумерация начинается с нуля. • [out] `read` – размер считанных данных в байтах. • [out] `data` – последовательность, содержащая считанные данные.	Нет.

	• [out] `rc` – код возврата.	
`GetSize`	Назначение Позволяет получить размер файла. Параметры • [in] `handle` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует открытый файл. • [out] `size` – размер файла в байтах. • [out] `rc` – код возврата.	Нет.
`GetId`	Назначение Позволяет получить уникальный идентификатор файла. Параметры • [in] `handle` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует открытый файл. • [out] `id` – уникальный идентификатор файла. • [out] `rc` – код возврата.	Нет.
`Count`	Назначение Позволяет получить число файлов в файловой системе. Параметры • [out] `count` – число файлов в файловой системе. • [out] `rc` – код возврата.	Нет.
`GetInfo`	Назначение Позволяет получить имя и уникальный идентификатор файла по индексу файла. Параметры • [in] `index` – индекс файла. Нумерация начинается с нуля. • [in] `nameLenMax` – размер буфера для сохранения имени файла.	Нет.

	• [out] `name` – имя файла. • [out] `id` – уникальный идентификатор файла. • [out] `rc` – код возврата.	
`GetFsSize`	Назначение Позволяет получить размер файловой системы. Параметры • [out] `fsSize` – размер файловой системы в байтах. • [out] `rc` – код возврата.	Нет.

Методы службы fs.FSUnsafe (интерфейс kl.core.FSUnsafe)

Метод	Назначение и параметры метода	Потенциальная опасность метода
`Change`	Назначение Меняет образ файловой системы. Вместо образа ROMFS, созданного при сборке решения, будет использоваться другой образ ROMFS, загруженный в память процесса. Параметры • [in] `base` – указатель на образ файловой системы. • [in] `size` – размер образа файловой системы в байтах. • [out] `rc` – код возврата.	Позволяет выполнить следующие действия: • Использовать образ ROMFS, содержащий произвольные программы и данные. • Получить доступ на чтение к некоторым объектам ядра.

Служба времени

Служба предназначена для установки системного времени.

Сведения о методах службы приведены в таблице ниже.

Методы службы time.Time (интерфейс kl.core.Time)

Метод	Назначение и параметры метода	Потенциальная опасность метода
`SetSystemTime`	Назначение Устанавливает системное время.	Позволяет установить системное время.

	Параметры • [in] `secs` – время, прошедшее с 1 января 1970 года, в секундах. • [in] `nsecs` – дополнительное время в наносекундах, которое складывается со временем, заданным через параметр `secs`. • [out] `rc` – код возврата.	
`SetSystemTimeAdj`	Назначение Запускает постепенную корректировку системного времени. Параметры • [in] `adj` – структура, содержащая интервал времени, на который нужно скорректировать системное время (*sec*10^9+nsec* наносекунд). • [in] `slew` – скорость корректировки системного времени (микросекунд в секунду). • [out] `prev` – структура, содержащая интервал времени, отражающий, на какое значение оставалось скорректировать системное время, чтобы предыдущая постепенная корректировка была полностью завершена (*sec*10^9+nsec* наносекунд). • [out] `rc` – код возврата.	Позволяет изменить системное время.
`GetSystemTimeAdj`	Назначение Позволяет получить интервал времени, отражающий, на какое значение остается скорректировать системное время, чтобы постепенная корректировка была полностью завершена. Параметры • [out] `adj` – структура, содержащая интервал времени, отражающий, на какое значение остается скорректировать системное время, чтобы постепенная корректировка была полностью завершена (*sec*10^9+nsec* наносекунд). • [out] `rc` – код возврата.	Нет.

Служба слоя аппаратных абстракций

Служба предназначена для получения значений параметров HAL, работы с привилегированными регистрами, очистки кеша процессора, выполнения диагностического вывода, а также получения случайных чисел, сгенерированных аппаратно.

Сведения о методах службы приведены в таблице ниже.

Методы службы hal.HAL (интерфейс kl.core.HAL)

Метод	Назначение и параметры метода	Потенциальная опасность метода
`GetEnv`	Назначение Позволяет получить значение параметра HAL. Параметры • [in] `name` – имя параметра. • [out] `value` – значение параметра. • [out] `rc` – код возврата.	Позволяет получить значения параметров HAL, которые могут представлять собой критические сведения о системе.
`GetPrivReg`	Назначение Позволяет получить значение привилегированного регистра. Параметры • [in] `reg` – имя регистра. • [out] `val` – значение регистра. • [out] `rc` – код возврата.	Позволяет организовать канал передачи данных с процессом, который имеет доступ к методу `SetPrivReg` или `SetPrivRegRange`. Рекомендуется контролировать имя регистра.
`SetPrivReg`	Назначение Задает значение привилегированного регистра. Параметры • [in] `reg` – имя регистра. • [in] `val` – значение регистра. • [out] `rc` – код возврата.	Позволяет выполнить следующие действия: • Задать значение привилегированного регистра. • Организовать канал передачи данных с процессом, который имеет доступ к методу `GetPrivReg` или `GetPrivRegRange`. Рекомендуется контролировать имя регистра.
`GetPrivRegRange`	Назначение Позволяет получить значение привилегированного регистра. Параметры • [in] `regRange` – имя диапазона регистров.	Позволяет организовать канал передачи данных с процессом, который имеет доступ к методу `SetPrivReg` или `SetPrivRegRange`. Рекомендуется контролировать имя диапазона регистров и смещение регистра в этом диапазоне.

	• [in] `offset` – смещение регистра в диапазоне регистров. • [out] `val` – значение регистра. • [out] `rc` – код возврата.	
`SetPrivRegRange`	Назначение Задает значение привилегированного регистра. Параметры • [in] `regRange` – имя диапазона регистров. • [in] `offset` – смещение регистра в диапазоне регистров. • [in] `val` – значение регистра. • [out] `rc` – код возврата.	Позволяет выполнить следующие действия: • Задать значение привилегированного регистра. • Организовать канал передачи данных с процессом, который имеет доступ к методу `GetPrivReg` или `GetPrivRegRange`. Рекомендуется контролировать имя диапазона регистров и смещение регистра в этом диапазоне.
`FlushCache`	Назначение Очищает кеш процессора. Параметры • [in] `type` – значение, задающее тип кеша (кеш данных, кеш инструкций, кеш данных и кеш инструкций совместно). • [in] `va` – базовый адрес региона виртуальной памяти. Кеш, соответствующий этому региону, очищается. • [in] `size` – размер региона виртуальной памяти. Кеш, соответствующий этому региону, очищается. • [out] `rc` – код возврата.	Позволяет очистить кеш процессора.
`DebugWrite`	Назначение	Позволяет заполнить диагностический вывод фиктивными данными (например, неинформативными).

	Помещает данные в диагностический вывод, который записывается, например, в порт COM или USB (версии 3.0 или более поздней, с поддержкой DbC). Параметры • [in] `data` – последовательность, содержащая данные для помещения в диагностический вывод. • [out] `rc` – код возврата.	
`GetEntropy`	Назначение Позволяет получить случайные числа, сгенерированные аппаратно. Параметры • [out] `buffer` – последовательность, содержащая случайные байтовые значения. • [in] `size` – число случайных байтовых значений. • [out] `rc` – код возврата.	Позволяет создать нагрузку на аппаратный генератор случайных чисел частыми вызовами метода, чтобы другие процессы не могли получить случайные числа с использованием этого генератора.

Служба управления контроллером XHCI

Служба предназначена для выключения и повторного включения отладочного режима контроллера XHCI (с поддержкой DbC) при его перезагрузке.

Сведения о методах службы приведены в таблице ниже.

Методы службы xhcidbg.XHCIDBG (интерфейс kl.core.XHCIDBG)

Метод	Назначение и параметры метода	Потенциальная опасность метода
`Start`	Назначение Включает отладочный режим контроллера XHCI. Параметры	Позволяет настроить контроллер XHCI, чтобы диагностический вывод выполнялся через порт USB (версии 3.0 или более поздней).

	• [out] `rc` – код возврата.	
`Stop`	Назначение Выключает отладочный режим контроллера XHCI. Параметры • [out] `rc` – код возврата.	Позволяет настроить контроллер XHCI, чтобы диагностический вывод не выполнялся через порт USB (версии 3.0 или более поздней).

Служба аудита

Служба предназначена для чтения сообщений из журналов ядра KasperskyOS. Этих журналов два: `kss` и `core`. Журнал `kss` содержит данные аудита безопасности. Журнал `core` содержит диагностический вывод. (Диагностический вывод включает как вывод ядра, так и вывод программ.)

Сведения о методах службы приведены в таблице ниже.

Методы службы audit.Audit (интерфейс kl.core.Audit)

Метод	Назначение и параметры метода	Потенциальная опасность метода
`Open`	Назначение Открывает журнал ядра для чтения. Параметры • [in] `name` – имя журнала ядра (`kss` или `core`). • [out] `handle` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует журнал ядра. • [out] `rc` – код возврата.	Нет.
`Close`	Назначение Закрывает журнал ядра. Параметры • [in] `handle` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует журнал ядра.	Нет.

	• [out] `rc` – код возврата.	
`Read`	Назначение Позволяет получить сообщение из журнала ядра. Параметры • [in] `handle` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует журнал ядра. • [out] `msg` – последовательность, содержащая сообщение. • [out] `outDropMsgs` – число сообщений, не попавших в журнал ядра из-за переполнения буфера, в котором этот журнал хранится. • [out] `rc` – код возврата.	Позволяет извлечь сообщения из журнала ядра, чтобы их не получил другой процесс.

Служба профилирования

Служба предназначена для профилирования и сбора покрытия кода, а также для получения значений счетчиков производительности.

Сведения о методах службы приведены в таблице ниже.

Методы службы profiler.Profiler (интерфейс kl.core.Profiler)

Метод	Назначение и параметры метода	Потенциальная опасность метода
`GetCoverageData`	Назначение Позволяет получить сведения о покрытии кода. Параметры • [in] `index` – индекс для перечисления объектных файлов, содержащих инструментированный код для получения сведений о покрытии. Нумерация начинается с нуля. • [out] `buf` – последовательность, содержащая сведения о покрытии кода объектного файла (в формате `gcda`). • [out] `size` – размер данных, содержащих сведения о покрытии кода объектного файла, в байтах. • [out] `name` – имя файла `*.gcda`, назначенное при компиляции.	Нет.

	• [out] `rc` – код возврата.	
`FlushGcov`	Назначение Выводит сведения о покрытии кода в формате `gcda` через UART. Параметры • [out] `rc` – код возврата.	Нет.
`FlushGcovFile`	Назначение Выводит сведения о покрытии кода в формате `gcda` через UART. Параметры • [in] `name` – имя файла `*.gcda`, назначенное при компиляции. • [in] `buf` – указатель на буфер, содержащий сведения о покрытии кода в формате `gcda`. • [in] `size` – размер данных, содержащих сведения о покрытии кода. • [out] `rc` – код возврата.	Нет.
`GetCounters`	Назначение Позволяет получить значения счетчиков производительности. Параметры • [in] `prefix` – префикс для имен счетчиков производительности. • [in] `names` – последовательность, содержащая имена счетчиков производительности. • [out] `values` – последовательность, содержащая значения счетчиков производительности. • [out] `rc` – код возврата.	Нет.
`ObjectGetStat`	Назначение Позволяет получить значения счетчиков производительности для системного ресурса (процесса или потока исполнения). Параметры	Нет.

	• [in] `handle` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует системный ресурс. • [in] `names` – последовательность, содержащая имена счетчиков производительности. • [out] `values` – последовательность, содержащая значения счетчиков производительности. • [out] `rc` – код возврата.	
`SamplingStart`	Назначение Запускает семплирующее профилирование кода. Результатом семплирующего профилирования является статистика исполнения кода, которая отражает длительность исполнения участков кода. Параметры • [in] `conf` – флаги, задающие параметры профилирования. • [in] `cpus` – значение, задающее процессоры (вычислительные ядра) для профилирования. • [in] `contSize` – размер контейнера для сохранения данных, содержащих статистику исполнения кода, полученную в результате профилирования, в байтах. Контейнер создается в памяти ядра автоматически. • [in] `interval` – фиктивный параметр. • [out] `rc` – код возврата.	Нет.
`SamplingStop`	Назначение Останавливает семплирующее профилирование кода. Параметры • [out] `rc` – код возврата.	Нет.
`SamplingRead`	Назначение Позволяет получить данные, содержащие статистику исполнения кода, полученную в результате семплирующего профилирования. Параметры	Позволяет получить адреса и имена функций других процессов.

	• [in] `unsafeBuffer` – указатель на буфер для сохранения контейнера, содержащего статистику исполнения кода, полученную в результате профилирования. • [in] `size` – размер буфера, указатель на который задан через параметр `unsafeBuffer`. • [out] `realSize` – размер сохраненного контейнера. • [in] `timeout` – время ожидания заполнения контейнера в миллисекундах. • [out] `rc` – код возврата.	
`SamplingAddPidToList`	Назначение Добавляет процесс в список профилируемых. Параметры • [in] `pid` – идентификатор процесса (PID). • [out] `rc` – код возврата.	Нет.
`SamplingClearPidList`	Назначение Очищает список профилируемых процессов. Параметры • [out] `rc` – код возврата.	Нет.
`LoadSegInfo`	Назначение Сохраняет в ядре сведения о загружаемом сегменте ELF-образа. (Это требуется, чтобы статистика исполнения кода, полученная в результате семплирующего профилирования, содержала дополнительную информацию, которая позволяет ассоциировать эту статистику с исходным кодом.) Параметры • [in] `task` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует процесс. • [in] `addr` – адрес сегмента в памяти процесса. • [in] `size` – размер сегмента в байтах.	Позволяет исчерпать память ядра, создавая в ней множество объектов.

	• [in] `offset` – смещение сегмента в файле ELF в байтах. • [in] `flags` – флаги, задающие права доступа к сегменту. • [in] `buildId` – идентификатор сборки. Компоновщик записывает этот идентификатор в файл ELF. • [out] `rc` – код возврата.	
`UnloadSegInfo`	Назначение Удаляет сведения о загружаемом сегменте ELF-образа, сохраненные в ядре методом `LoadSegInfo`. Параметры • [in] `task` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует процесс. • [in] `addr` – адрес сегмента в памяти процесса. • [in] `size` – размер сегмента в байтах. • [out] `rc` – код возврата.	Нет.
`KcovAlloc`	Назначение Выделяет ресурсы, требуемые для сбора покрытия кода ядра, который осуществляется при обработке системных вызовов, выполняемых вызывающим процессом. Параметры • [in] `numThreads` – максимальное число потоков исполнения, для которых будет выполняться сбор покрытия. • [in] `maxPoints` – максимальное число точек покрытия для одного потока исполнения. • [out] `rc` – код возврата.	Позволяет исчерпать оперативную память.
`KcovFree`	Назначение Освобождает ресурсы, требуемые для сбора покрытия кода ядра, который осуществляется при обработке системных вызовов, выполняемых вызывающим процессом. Параметры	Нет.

	• [out] `rc` – код возврата.	
`KcovStart`	Назначение Запускает сбор покрытия кода ядра, который осуществляется при обработке системных вызовов, выполняемых вызывающим потоком исполнения. Параметры • [out] `rc` – код возврата.	Нет.
`KcovStop`	Назначение Останавливает сбор покрытия кода ядра, который осуществляется при обработке системных вызовов, выполняемых вызывающим потоком исполнения. Также позволяет получить сведения о покрытии кода ядра. Параметры • [in] `points` – указатель на буфер для сохранения сведений о покрытии кода ядра. • [in] `maxPoints` – максимальное число точек покрытия, которые можно сохранить в буфере, заданном через параметр `points`. • [out] `numPoints` – фактическое число точек покрытия, сохраненных в буфере, заданном через параметр `points`. • [out] `rc` – код возврата.	Нет.

Служба управления изоляцией памяти для ввода-вывода

Служба предназначена для управления изоляцией регионов физической памяти, используемых устройствами на шине PCIe для DMA. (Изоляция обеспечивается IOMMU.)

Сведения о методах службы приведены в таблице ниже.

Методы службы iommu.IOMMU (интерфейс kl.core.IOMMU)

Метод	Назначение и параметры метода	Потенциальная опасность метода
`Attach`	Назначение Прикрепляет устройство на шине PCIe к домену IOMMU, ассоциированному с вызывающим процессом.	Позволяет прикрепить устройство на шине PICIe, управляемое другим процессом, к домену IOMMU, ассоциированному с вызывающим процессом, что приведет к неработоспособности устройства. Рекомендуется контролировать адрес устройства на шине PCIe.

	Параметры • [in] `bdf` – адрес устройства на шине PCIe в формате BDF. • [out] `rc` – код возврата.	
`Detach`	Назначение Открепляет устройство на шине PCIe от домена IOMMU, ассоциированного с вызывающим процессом. Параметры • [in] `bdf` – адрес устройства на шине PCIe в формате BDF. • [out] `rc` – код возврата.	Нет.

Служба соединений

Служба предназначена для динамического создания IPC-каналов.

Сведения о методах службы приведены в таблице ниже.

Методы службы cm.CM (интерфейс kl.core.CM)

Метод	Назначение и параметры метода	Потенциальная опасность метода
`Connect`	Назначение Выполняет запрос на создание IPC-канала с сервером для использования заданной службы. Параметры • [in] `server` – имя сервера. • [in] `service` – квалифицированное имя службы. • [in] `msecs` – время ожидания выполнения запроса в миллисекундах. • [out] `handle` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор является клиентским IPC-дескриптором. • [out] `id` – идентификатор службы (RIID).	Позволяет создать нагрузку на сервер, отправляя множество запросов на создание IPC-канала.

	• [out] `rc` – код возврата.	
`Listen`	Назначение Позволяет получить запрос клиента на создание IPC-канала для использования службы. Параметры • [in] `filter` – фиктивный параметр. • [in] `msecs` – время ожидания запроса клиента в миллисекундах. • [out] `client` – имя клиента. • [out] `service` – квалифицированное имя службы. • [out] `rc` – код возврата.	Нет.
`Drop`	Назначение Отклоняет запрос клиента на создание IPC-канала для использования заданной службы. Параметры • [in] `client` – имя клиента. • [in] `service` – квалифицированное имя службы. • [out] `rc` – код возврата.	Нет.
`Accept`	Назначение Принимает запрос клиента на создание IPC-канала для использования заданной службы. Параметры • [in] `client` – имя клиента. • [in] `service` – квалифицированное имя службы. • [in] `id` – идентификатор службы. • [in] `listener` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор является слушающим дескриптором. • [out] `handle` – значение, двоичное представление которого состоит из нескольких полей, включая поле	Нет.

	дескриптора и поле маски прав дескриптора. Дескриптор является серверным IPC-дескриптором. • [out] `rc` – код возврата.	

Служба управления электропитанием

Служба предназначена для изменения режима электропитания компьютера (например, выключения, перезагрузки), а также для включения и выключения процессоров (вычислительных ядер).

Сведения о методах службы приведены в таблице ниже.

Методы службы pm.PM (интерфейс kl.core.PM)

Метод	Назначение и параметры метода	Потенциальная опасность метода
`Request`	Назначение Выполняет запрос на изменение режима электропитания компьютера. Параметры • [in] `request` – значение, задающее требуемый режим электропитания компьютера. • [out] `rc` – код возврата.	Позволяет изменить режим электропитания компьютера.
`SetCpusOnline`	Назначение Выполняет запрос на включение и/или выключение процессоров. Параметры • [in] `request` – значение, задающее множество процессоров в активном состоянии. • [in] `timeout` – время ожидания выполнения запроса в миллисекундах. • [out] `rc` – код возврата.	Позволяет выключить и включить процессоры.
`GetCpusOnline`	Назначение Позволяет получить сведения о том, какие процессоры находятся в активном состоянии. Параметры • [out] `online` – значение, отражающее множество процессоров в активном состоянии.	Нет.

	• [out] `rc` – код возврата.	

Служба уведомлений

Служба предназначена для работы с уведомлениями о событиях, происходящих с ресурсами.

Сведения о методах службы приведены в таблице ниже.

Методы службы notice.Notice (интерфейс kl.core.Notice)

Метод	Назначение и параметры метода	Потенциальная опасность метода
`Create`	Назначение Создает приемник уведомлений. Параметры • [out] `notify` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует приемник уведомлений. • [out] `rc` – код возврата.	Позволяет исчерпать память ядра, создавая в ней множество объектов.
`SubscribeToObject`	Назначение Добавляет запись вида "ресурс – маска событий" в приемник уведомлений, чтобы он получал уведомления о событиях, которые происходят с заданным ресурсом и соответствуют заданной маске событий. Параметры • [in] `notify` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует приемник уведомлений. • [in] `object` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует ресурс. • [in] `evMask` – маска событий. • [in] `evId` – идентификатор записи вида "ресурс – маска событий". • [out] `rc` – код возврата.	Позволяет исчерпать память ядра, создавая в ней множество объектов.

`UnsubscribeFromEvent`	Назначение Удаляет записи вида "ресурс – маска событий" с заданным идентификатором из приемника уведомлений, чтобы он не получал уведомления о событиях, соответствующих этим записям. Параметры • [in] `notify` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует приемник уведомлений. • [in] `evId` – идентификатор записи вида "ресурс – маска событий". • [out] `rc` – код возврата.	Нет.
`UnsubscribeFromObject`	Назначение Удаляет записи вида "ресурс – маска событий", соответствующие заданному ресурсу, из приемника уведомлений, чтобы он не получал уведомления о событиях, соответствующих этим записям. Параметры • [in] `notify` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует приемник уведомлений. • [in] `object` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует ресурс. • [out] `rc` – код возврата.	Нет.
`GetEvent`	Назначение Извлекает уведомления из приемника. Параметры • [in] `notify` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует приемник уведомлений. • [in] `mdelay` – время ожидания появления уведомлений в приемнике в миллисекундах. • [out] `events` – последовательность уведомлений, которые представляют собой структуры,	Нет.

	содержащие идентификатор записи вида "ресурс – маска событий" и маску событий, произошедших с ресурсом. • [out] `rc` – код возврата.	
`DropAndWake`	Назначение Удаляет все записи вида "ресурс – маска событий"из заданного приемника уведомлений; возобновляет исполнение всех потоков, ожидающих появления уведомлений в заданном приемнике; опционально запрещает добавление записей вида "ресурс – маска событий" в заданный приемник уведомлений. Параметры • [in] `notify` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует приемник уведомлений. • [in] `finish` – значение, определяющее будет ли запрещено добавление записей вида "ресурс – маска событий" (`0` – не будет запрещено, `1` – будет запрещено). • [out] `rc` – код возврата.	Нет.
`SetObjectEvent`	Назначение Сигнализирует, что события из заданной маски событий произошли с заданным пользовательским ресурсом. Параметры • [in] `object` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует пользовательский ресурс. • [in] `evMask` – маска событий, о которых требуется сигнализировать. • [out] `rc` – код возврата.	Нет.

Служба гипервизора

Служба предназначена для работы с гипервизором.

Методы службы `hypervisor.Hypervisor` (интерфейс `kl.core.Hypervisor`) являются потенциально опасными. Доступ к этим методам можно разрешать только специальной программе `vmapp`.

Службы доверенной среды исполнения

Службы предназначены для передачи данных между *доверенной средой исполнения* (англ. Trusted Execution Environment, TEE) и *общей средой исполнения* (англ. Rich Execution Environment, REE), а также для получения доступа к физической памяти REE из TEE.

Сведения о методах служб приведены в таблицах ниже.

Методы службы tee.TEE (интерфейс kl.core.TEE)

Метод	Назначение и параметры метода	Потенциальная опасность метода
`Dispatch`	Назначение Отправляет и принимает сообщения, передающиеся между TEE и REE. Метод используется как в TEE, так и в REE. Параметры • [in] `msgIn` – структура, содержащая запрос для TEE (при вызове метода в REE) или ответ для REE (при вызове метода в TEE). • [out] `msgOut` – структура, содержащая ответ от TEE (при вызове метода в REE) или запрос от REE (при вызове метода в TEE). • [out] `rc` – код возврата.	Позволяет процессу в REE получить ответ от TEE на запрос другого процесса в REE.
`FreeToken`	Назначение Освобождает значения уникальных идентификаторов сообщений, передающихся между TEE и REE. (Эти значения требуется освободить, чтобы они стали доступными для повторного использования.) Метод используется в REE. Параметры • [in] `token` – значение уникального идентификатора сообщения. • [out] `rc` – код возврата.	Позволяет освободить значения, используемые другими процессами в REE в качестве уникальных идентификаторов сообщений, передающихся между TEE и REE.

Методы службы tee.TEEVMM (интерфейс kl.core.TEEVMM)

Метод	Назначение и параметры метода	Потенциальная опасность метода

`MdlAllocate`	Назначение Создает заготовку буфера MDL для последующего добавления в нее физической памяти из REE. Метод используется в TEE. Параметры • [in] `size` – размер буфера MDL в байтах. • [in] `prot` – флаги, задающие права доступа к буферу MDL. • [out] `handle` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует буфер MDL. • [out] `rc` – код возврата.	Позволяет исчерпать память ядра, создавая в ней множество объектов.
`MdlAddFrame`	Назначение Добавляет регион физической памяти REE в заготовку буфера MDL, созданную методом `MdlAllocate`. Метод используется в TEE. Параметры • [in] `handle` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует буфер MDL. • [in] `pa` – базовый адрес региона физической памяти. • [in] `pages` – размер региона физической памяти в страницах памяти. • [out] `rc` – код возврата.	Позволяет получить доступ к произвольному региону физической памяти REE из TEE.

Служба прерывания IPC

Служба предназначена для прерывания блокирующих системных вызовов `Call()` и `Recv()`. (Это может потребоваться, например, для корректного завершения процесса.)

Сведения о методах службы приведены в таблице ниже.

Методы службы ipc.IPC (интерфейс kl.core.IPC)

Метод	Назначение и параметры метода	Потенциальная опасность

		метода
`CreateSyncObject`	Назначение Создает объект синхронизации IPC. Объект синхронизации IPC используется для прерывания блокирующих системных вызовов `Call()` и `Recv()` в потоках вызывающего процесса. `Call()` может быть прерван только тогда, когда он ожидает вызова `Recv()` сервером. `Recv()` может быть прерван только тогда, когда он ожидает получения IPC-запроса от клиента. Дескриптор объекта синхронизации IPC не может быть передан другому процессу, так как в маске прав этого дескриптора не установлен необходимый для этого флаг. Параметры • [out] `syncHandle` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует объект синхронизации IPC. • [out] `rc` – код возврата.	Позволяет исчерпать память ядра, создавая в ней множество объектов.
`SetInterrupt`	Назначение Переводит заданный объект синхронизации IPC в состояние, при котором системные вызовы `Call()` и `Recv()` прерываются. Параметры • [in] `syncHandle` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует объект синхронизации IPC. • [out] `rc` – код возврата.	Нет.
`ClearInterrupt`	Назначение Переводит заданный объект синхронизации IPC в состояние, при котором системные вызовы `Call()` и `Recv()` не прерываются. Параметры • [in] `syncHandle` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует объект синхронизации IPC. • [out] `rc` – код возврата.	Нет.

Служба управления частотой процессоров

Служба предназначена для изменения частоты процессоров (вычислительных ядер).

Сведения о методах службы приведены в таблице ниже.

Методы службы cpufreq.CpuFreq (интерфейс kl.core.CpuFreq)

Метод	Назначение и параметры метода	Потенциальная опасность метода
`GetLayout`	Назначение Позволяет получить сведения о процессорных группах. В сведениях о процессорных группах перечислены существующие процессорные группы с указанием возможных значений параметра производительности для каждой из них. Этим параметром является комбинация соответствующих друг другу частоты и напряжения (англ. Operating Performance Point, OPP). Частота приводится в кГц, напряжение приводится в мкВ. Параметры • [out] `layout` – последовательность, содержащая сведения о процессорных группах. • [out] `rc` – код возврата.	Нет.
`GetCurOppId`	Назначение Позволяет получить индекс текущего OPP для заданной процессорной группы. Параметры • [in] `cpuGroupId` – индекс процессорной группы. Нумерация начинается с нуля. • [out] `oppId` – индекс текущего OPP. Нумерация начинается с нуля. • [out] `rc` – код возврата.	Нет.
`SetOppId`	Назначение Устанавливает заданный OPP для заданной процессорной группы. Параметры • [in] `GroupId` – индекс процессорной группы. Нумерация начинается с нуля.	Позволяет изменить частоту процессорной группы.

- [in] `oppId` – индекс OPP. Нумерация начинается с нуля.
- [out] `rc` – код возврата.

Использование системных программ Klog и KlogStorage для выполнения аудита безопасности

Для выполнения аудита безопасности системная программа `Klog` получает данные аудита от ядра KasperskyOS, используя библиотеку `libkos`, декодирует эти данные и передает через IPC системной программе `KlogStorage`, выступающей в этом IPC-взаимодействии в качестве сервера. Программа `KlogStorage` направляет данные аудита в стандартный вывод (или стандартный вывод ошибок) либо сохраняет в файл, используя VFS. Также программа `KlogStorage` может передавать данные аудита, записанные в файл, другим программам через IPC.

Исполняемые файлы программ `Klog` и `KlogStorage` не поставляются в составе KasperskyOS SDK. Их нужно создать на основе поставляемых статических библиотек.

Пример включения в решение системной программы Klog

Исходный код программы

einit/src/klog_entity.c

```
#include <klog/system_audit.h>
#include <klog_storage/client.h>
#include <ping/KlogEntity.edl.h>

int main(int argc, char *argv[])
{
    /* В результате вызова этой функции будет создан поток исполнения,
     * который получает данные аудита от ядра, декодирует их и передает
     * через IPC программе KlogStorage.
     * (Константа ping_KlogEntity_klog_audit_iid определена в заголовочном
     * файле KlogEntity.edl.h, который содержит автоматически сгенерированный
     * транспортный код.) */
    return klog_system_audit_run(KLOG_SERVER_CONNECTION_ID ":"
                                 " KLOG_STORAGE_SERVER_CONNECTION_ID,
                                 ping_KlogEntity_klog_audit_iid);
}
```

Сборка программы

einit/CMakeLists.txt

```
...
# Импорт библиотек Klog из состава
# KasperskyOS SDK
find_package (klog REQUIRED)
```

```
include_directories (${klog_INCLUDE})

# Генерация транспортного кода на основе формальной спецификации
# программы Klog
nk_build_edl_files (klog_edl_files
                    NK_MODULE "ping"
                    # Файл KlogEntity.edl и другие файлы
                    # формальной спецификации программы Klog
                    # поставляются в составе KasperskyOS SDK.
                    EDL "${RESOURCES}/edl/KlogEntity.edl")

# Создание исполняемого файла программы Klog для аппаратной платформы
add_executable (KlogEntityHw "src/klog_entity.c")
target_link_libraries (KlogEntityHw ${klog_SYSTEM_AUDIT_LIB})
add_dependencies (KlogEntityHw klog_edl_files)

# Создание исполняемого файла программы Klog для QEMU.
# (Идентично созданию исполняемого файла программы Klog для
# аппаратной платформы, за исключением имени цели сборки.
# Требуется две цели сборки исполняемого файла программы
# Klog с разными именами, поскольку в параметре KLOG_ENTITY
# CMake-команд build_kos_hw_image() и build_kos_qemu_image()
# нужно указать разные цели сборки.)
add_executable (KlogEntityQemu "src/klog_entity.c")
target_link_libraries (KlogEntityQemu ${klog_SYSTEM_AUDIT_LIB})
add_dependencies (KlogEntityQemu klog_edl_files)

# Программу Klog не нужно указывать вместе с другими программами
# для включения в образ решения. Чтобы включить программу Klog
# в решение, нужно задать имя цели сборки исполняемого файла этой
# программы через параметр KLOG_ENTITY CMake-команд
# build_kos_hw_image() и build_kos_qemu_image().
set (ENTITIES Client Server KlogStorageEntity FileVfs)
...
# Переменная INIT_KlogEntity_PATH используется в файле init.yaml.in,
# чтобы задать имя исполняемого файла программы Klog. (Исполняемые
# файлы программы Klog для QEMU и для аппаратной платформы имеют
# разные имена, которые по умолчанию совпадают с именами целей сборки
# этих файлов.)
set (INIT_KlogEntity_PATH "KlogEntityHw")

# Нужно задать параметр KLOG_ENTITY
build_kos_hw_image (kos-image
                    EINIT_ENTITY EinitHw
                    ...
                    KLOG_ENTITY KlogEntityHw
                    IMAGE_FILES ${ENTITIES})

# Переменная INIT_KlogEntity_PATH используется в файле init.yaml.in,
# чтобы задать имя исполняемого файла программы Klog. (Исполняемые
# файлы программы Klog для QEMU и для аппаратной платформы имеют
# разные имена, которые по умолчанию совпадают с именами целей сборки
# этих файлов.)
set (INIT_KlogEntity_PATH "KlogEntityQemu")

# Нужно задать параметр KLOG_ENTITY
build_kos_qemu_image (kos-qemu-image
                      EINIT_ENTITY EinitQemu
                      ...
                      KLOG_ENTITY KlogEntityQemu
                      IMAGE_FILES ${ENTITIES})
```

Словарь процесса программы в шаблоне init-описания

einit/src/init.yaml.in

```
...
- name: ping.KlogEntity
  # Переменная INIT_KlogEntity_PATH определена в файле einit/CMakeLists.txt.
  path: @INIT_KlogEntity_PATH@
  connections:
  - target: ping.KlogStorageEntity
    id: {var: KLOG_STORAGE_SERVER_CONNECTION_ID, include: klog_storage/client.h}
...
```

Описание политики для программы

einit/src/security.psl.in

```
...
use nk.base._
...
use EDL kl.core.Core
...
use EDL ping.KlogEntity
use EDL ping.KlogStorageEntity
...
use audit_profile._
use core._
...
/* Взаимодействие с программой KlogStorage */

request dst=ping.KlogStorageEntity {
    match endpoint=klogStorage.storage {
        match method=write {
            match src=ping.KlogEntity { grant () }
        }
    }
}

response src=ping.KlogStorageEntity {
    match endpoint=klogStorage.storage {
        match method=write {
            match dst=ping.KlogEntity { grant () }
        }
    }
}

error src=ping.KlogStorageEntity {
    match endpoint=klogStorage.storage {
        match method=write {
            match dst=ping.KlogEntity { grant () }
        }
    }
}
...
```

einit/src/core.psl

```
...
/* Взаимодействие с ядром */

request dst=kl.core.Core {
    match endpoint=sync.Sync {
        match method=Wake {
            ...
            match src=ping.KlogEntity { grant () }
            ...
            }
        match method=Wait {
           ...
            match src=ping.KlogEntity { grant () }
          ...
            }
    }
    match endpoint=task.Task {
        match method=FreeSelfEnv {
            ...
            match src=ping.KlogEntity { grant () }
            ...
            }
        match method=GetPath {
            ...
            match src=ping.KlogEntity { grant () }
            ...
            }
        match method=GetName {
            ...
            match src=ping.KlogEntity { grant () }
            ...
            }
        match method=Exit {
            ...
            match src=ping.KlogEntity { grant () }
            ...
            }
    }
    match endpoint=vmm.VMM {
        match method=Allocate {
            ...
            match src=ping.KlogEntity { grant () }
            ...
            }
        match method=Commit {
            ...
            match src=ping.KlogEntity { grant () }
            ...
            }
        match method=Protect {
            ...
            match src=ping.KlogEntity { grant () }
            ...
            }
        match method=Free {
            ...
            match src=ping.KlogEntity { grant () }
            ...
            }
    }
    match endpoint=thread.Thread {
```

```
        match method=SetTls {
            ...
            match src=ping.KlogEntity { grant () }
            ...
            }
        match method=Create {
            ...
            match src=ping.KlogEntity { grant () }
            ...
            }
        match method=Resume {
            ...
            match src=ping.KlogEntity { grant () }
            ...
            }
        match method=Attach {
            ...
            match src=ping.KlogEntity { grant () }
            ...
            }
        match method=Exit {
            ...
            match src=ping.KlogEntity { grant () }
            ...
            }
        match method=GetSchedPolicy {
            ...
            match src=ping.KlogEntity { grant () }
            ...
            }
        match method=SetSchedPolicy {
            ...
            match src=ping.KlogEntity { grant () }
            ...
            }
    }
    match endpoint=hal.HAL {
        match method=GetEntropy {
            ...
            match src=ping.KlogEntity { grant () }
            ...
            }
        match method=DebugWrite {
            ...
            match src=ping.KlogEntity { grant () }
            ...
            }
        match method=GetEnv {
            ...
            match src=ping.KlogEntity { grant () }
            ...
            }
    }
    match endpoint=handle.Handle {
        match method=Close {
            ...
            match src=ping.KlogEntity { grant () }
            ...
            }
    }
    match endpoint=audit.Audit {
        match src=ping.KlogEntity { grant () }
```

```
        }
}

response src=kl.core.Core {
    ...
    match dst=ping.KlogEntity { grant () }
    ...
}

error src=kl.core.Core {
    ...
    match dst=ping.KlogEntity { grant () }
    ...
}
...
```

Пример включения в решение системной программы KlogStorage, направляющей данные аудита в стандартный вывод ошибок

Исходный код программы

klog_storage/src/klog_storage_entity.c

```c
#include <klog_storage/server.h>
#include <ping/KlogStorageEntity.edl.h>
#include <stdio.h>

/* Определение типа данных для фиктивного контекста.
 * Требуется для определения функций, реализующих
 * интерфейсные методы, и настройки диспетчера. */
struct Context
{
    int some_data;
};

/* Определение функции, направляющей данные аудита в стандартный
 * вывод ошибок. (Параметр ctx не требуется использовать, но параметр
 * типа void* должен быть первым параметром в сигнатуре функции, чтобы
 * соответствовать типу указателя, который используется диспетчером
 * для вызова этой функции.) */
static int _write(struct Context *ctx, const struct kl_KlogStorage_Entry *entry)
{
    fprintf(stderr, "%s\n", entry->msg);
    return 0;
}

/* Определение фиктивной функции чтения данных аудита.
 * (Требуется для настройки диспетчера, чтобы избежать ошибки,
 * если будет вызван интерфейсный метод чтения данных аудита.) */
static int _read_range(struct Context *ctx, nk_uint64_t first_id,
nk_uint64_t last_id, struct kl_KlogStorage_Entry *entries)
{
    return 0;
}

/* Определение фиктивной функции чтения данных аудита.
```

```
 * (Требуется для настройки диспетчера, чтобы избежать ошибки,
 * если будет вызван интерфейсный метод чтения данных аудита.) */
static int _read(struct Context *ctx, nk_uint32_t num_entries,
struct kl_KlogStorage_Entry *entries)
{
    return 0;
}

int main(int argc, char *argv[])
{
    /* Объявление фиктивного контекста */
    static struct Context ctx;

    /* Настройка диспетчера, чтобы при получении от программы Klog
     * IPC-запросов с данными аудита была вызвана функция, направляющая
     * эти данные в стандартный вывод ошибок. (Функции чтения данных аудита
     * и контекст являются фиктивными. Однако можно создать собственные
     * реализации функций _write(), _read() и _read_range() для работы с
     * хранилищем данных аудита. В этом случае контекст может быть
     * использован для хранения состояния хранилища.) */
    struct kl_KlogStorage *iface =
    klog_storage_IKlog_storage_dispatcher(&ctx,
                                          (kl_KlogStorage_write_func)_write,
                                          (kl_KlogStorage_read_func)_read,

(kl_KlogStorage_read_range_func)_read_range);
    struct kl_KlogStorage_component *comp =klog_storage_storage_component(iface);

    /* В результате вызова этой функции будет запущен цикл обработки IPC-запросов.
     * (Константы ping_KlogStorageEntity_klogStorage_iidOffset и
     * ping_KlogStorageEntity_klogStorage_storage_iid определены в заголовочном файле
     * KlogStorageEntity.edl.h, который содержит автоматически сгенерированный
     * транспортный код.) */
    return klog_storage_run(KLOG_STORAGE_SERVER_CONNECTION_ID,
                            ping_KlogStorageEntity_klogStorage_iidOffset,
                            ping_KlogStorageEntity_klogStorage_storage_iid,
                            comp);
}
```

Сборка программы

klog_storage/CMakeLists.txt

```
# Импорт библиотек KlogStorage из состава
# KasperskyOS SDK
find_package (klog_storage REQUIRED)
include_directories (${klog_storage_INCLUDE})

# Генерация транспортного кода на основе формальной спецификации
# программы KlogStorage
nk_build_edl_files (klog_storage_edl_files
                    NK_MODULE "ping"
                    # Файл KlogStorageEntity.edl и другие файлы
                    # формальной спецификации программы KlogStorage
                    # поставляются в составе KasperskyOS SDK.
                    EDL "${RESOURCES}/edl/KlogStorageEntity.edl")

# Создание исполняемого файла программы KlogStorage
add_executable (KlogStorageEntity "src/klog_storage_entity.c")
```

```
target_link_libraries (KlogStorageEntity ${klog_storage_SERVER_LIB})
add_dependencies (KlogStorageEntity klog_edl_files klog_storage_edl_files)
```

Словарь процесса программы в шаблоне init-описания

einit/src/init.yaml.in

```
...
- name: ping.KlogStorageEntity
...
```

Описание политики для программы

einit/src/security.psl.in

```
...
use nk.base._
...
use EDL kl.core.Core
...
use EDL ping.KlogEntity
use EDL ping.KlogStorageEntity
...
use audit_profile._
use core._
...
/* Взаимодействие с программой Klog */

request dst=ping.KlogStorageEntity {
    match endpoint=klogStorage.storage {
        match method=write {
            match src=ping.KlogEntity { grant () }
        }
    }
}

response src=ping.KlogStorageEntity {
    match endpoint=klogStorage.storage {
        match method=write {
            match dst=ping.KlogEntity { grant () }
        }
    }
}

error src=ping.KlogStorageEntity {
    match endpoint=klogStorage.storage {
        match method=write {
            match dst=ping.KlogEntity { grant () }
        }
    }
}
...
```

einit/src/core.psl

```
...
/* Взаимодействие с ядром */

request dst=kl.core.Core {
    match endpoint=sync.Sync {
        match method=Wake {
            ...
            match src=ping.KlogStorageEntity { grant () }
            ...
            }
        match method=Wait {
            ...
            match src=ping.KlogStorageEntity { grant () }
            ...
            }
    }
    match endpoint=task.Task {
        match method=FreeSelfEnv {
            ...
            match src=ping.KlogStorageEntity { grant () }
            ...
            }
        match method=GetPath {
            ...
            match src=ping.KlogStorageEntity { grant () }
            ...
            }
        match method=GetName {
            ...
            match src=ping.KlogStorageEntity { grant () }
            ...
            }
        match method=Exit {
            ...
            match src=ping.KlogStorageEntity { grant () }
            ...
            }
    }
    match endpoint=vmm.VMM {
        match method=Allocate {
            ...
            match src=ping.KlogStorageEntity { grant () }
            ...
            }
        match method=Commit {
            ...
            match src=ping.KlogStorageEntity { grant () }
            ...
            }
        match method=Protect {
            ...
            match src=ping.KlogStorageEntity { grant () }
            ...
            }
        match method=Free {
            ...
            match src=ping.KlogStorageEntity { grant () }
            ...
            }
    }
    match endpoint=thread.Thread {
```

```
        match method=SetTls {
            ...
            match src=ping.KlogStorageEntity { grant () }
            ...
            }
        match method=Create {
            ...
            match src=ping.KlogStorageEntity { grant () }
            ...
            }
        match method=Resume {
            ...
            match src=ping.KlogStorageEntity { grant () }
            ...
            }
    }
    match endpoint=hal.HAL {
        match method=GetEntropy {
            ...
            match src=ping.KlogStorageEntity { grant () }
            ...
            }
        match method=DebugWrite {
            ...
            match src=ping.KlogStorageEntity { grant () }
            ...
            }
        match method=GetEnv {
            ...
            match src=ping.KlogStorageEntity { grant () }
            ...
            }
    }
    match endpoint=handle.Handle {
        match method=Close {
            ...
            match src=ping.KlogStorageEntity { grant () }
            ...
            }
    }
}

response src=kl.core.Core {
    ...
    match dst=ping.KlogStorageEntity { grant () }
    ...
}

error src=kl.core.Core {
    ...
    match dst=ping.KlogStorageEntity { grant () }
    ...
}
...
```

Пример включения в решение системной программы KlogStorage, выполняющей запись данных аудита в файл

Исходный код программы

klog_storage/src/klog_storage_entity.c

```
#include <klog_storage/server.h>
#include <klog_storage/file_storage.h>
#include <ping/KlogStorageEntity.edl.h>

int main(int argc, char *argv[])
{
    /* В результате вызова этой функции будет запущен цикл обработки IPC-запросов.
     * Данные аудита будут записываться в файл /etc/klog_storage.log, который может
     * вместить не более 100 записей. При полном заполнении файла предыдущие
     * записи будут заменяться новыми с начала файла. Если в последнем параметре
     * функции указано значение, отличное от 1, программа KlogStorage при запуске
     * откроет существующий файл и начнет запись данных аудита с той позиции,
     * которая была установлена в файле после предыдущей записи. Если в последнем
     * параметре функции указано значение 1, будет создан новый пустой файл.
     * (Константы ping_KlogStorageEntity_klogStorage_iidOffset и
     * ping_KlogStorageEntity_klogStorage_storage_iid определены в заголовочном
     * файле KlogStorageEntity.edl.h, который содержит автоматически сгенерированный
     * транспортный код.) */
    return klog_storage_file_storage_run(KLOG_STORAGE_SERVER_CONNECTION_ID,
                                         "/etc/klog_storage.log",
                                         ping_KlogStorageEntity_klogStorage_iidOffset,

ping_KlogStorageEntity_klogStorage_storage_iid,
                                         100,
                                         0);
}
```

Сборка программы

`CMake`-команды для сборки программы `KlogStorage`, выполняющей запись данных аудита в файл, отличаются от `CMake`-команд для сборки версии этой программы, направляющей данные аудита в стандартный вывод ошибок, следующим изменением:

klog_storage/CMakeLists.txt

```
...
# При создании исполняемого файла программы KlogStorage нужно
# выполнить компоновку с библиотекой klog_storage_file_storage.
target_link_libraries (KlogStorageEntity ${klog_storage_FILE_STORAGE_LIB})
...
```

Словарь процесса программы в шаблоне init-описания

einit/src/init.yaml.in

```
...
- name: ping.KlogStorageEntity
  connections:
  - target: file_vfs.FileVfs
    id: {var: _VFS_CONNECTION_ID, include: vfs/defs.h}
...
```

Описание политики безопасности для программы

Описание политики для программы `KlogStorage`, выполняющей запись данных аудита в файл, отличается от описания политики для версии этой программы, направляющей данные аудита в стандартный вывод ошибок, следующим дополнением:

einit/src/security.psl.in

```
...
use EDL file_vfs.FileVfs
...
use vfs._
...
```

einit/src/vfs.psl

```
...
/* Взаимодействие с программой VFS */

request dst=file_vfs.FileVfs {
    match src=ping.KlogStorageEntity { grant () }
}

response src=file_vfs.FileVfs {
    match dst=ping.KlogStorageEntity { grant () }
}

error src=file_vfs.FileVfs {
    match dst=ping.KlogStorageEntity { grant () }
}
...
```

Передача данных аудита другим программам

Чтобы передавать через IPC данные аудита, записанные в файл, программа `KlogStorage` предоставляет интерфейсные методы `read` и `readRange`, определенные в файле `sysroot-*-kos/include/kl/KlogStorage.idl` из состава KasperskyOS SDK.

Исполняемый файл программы, которой требуется получать данные аудита, должен быть скомпонован с клиентской библиотекой программы `KlogStorage`:

klog_reader/CMakeLists.txt

```
# Импорт библиотек KlogStorage из состава
# KasperskyOS SDK
find_package (klog_storage REQUIRED)
include_directories (${klog_storage_INCLUDE})
...
# Создание исполняемого файла программы, которой требуется
# получать данные аудита от программы KlogStorage.
add_executable (KlogReader "src/klog_reader.c")
target_link_libraries (KlogReader ${klog_storage_CLIENT_LIB})
...
```

Исходный код для получения данных аудита от программы `KlogStorage`:

klog_reader/src/klog_reader.c

```
#include <klog_storage/client.h>
...
int main(int argc, char *argv[])
{
...
    struct Klog_storage_ctx *storage =
    klog_storage_init(KLOG_STORAGE_SERVER_CONNECTION_ID);

    struct kl_KlogStorage_Entry first_entries[10], latest_entries [10];

    /* Чтение десяти первых записей */
    int f_count = klog_storage_read_range(klog_storage_IKlog_storage(storage),
                                          1,
                                          10,
                                          first_entries);

    /* Чтение десяти последних записей */
    int l_count = klog_storage_read(klog_storage_IKlog_storage(storage),
                                    10,
                                    latest_entries);
...
}
```

Паттерны безопасности при разработке под KasperskyOS

Каждое решение на базе KasperskyOS имеет определенные сценарии использования и предназначено для противодействия конкретным угрозам безопасности. Тем не менее, существуют типовые сценарии и угрозы, которые встречаются во многих решениях. Этот раздел описывает типовые риски и угрозы, а также содержит описание архитектурных паттернов, применение которых позволит повысить безопасность решения.

Паттерн (или шаблон) безопасности описывает конкретную повторяющуюся проблему безопасности, которая возникает в определенных известных контекстах, а также предлагает хорошо зарекомендовавшую себя общую схему решения такой проблемы безопасности. Паттерн это не законченный проект, который можно преобразовать непосредственно в код, а решение общей проблемы, встречающейся в различных проектах.

Система паттернов безопасности – это набор паттернов безопасности вместе с инструкциями по их реализации, сочетанию и практическому использованию в проектировании безопасных программных систем.

Паттерны безопасности решают проблемы безопасности на разных уровнях: начиная от паттернов архитектурного уровня, включающих высокоуровневый дизайн системы, и заканчивая паттернами уровня реализации, содержащими рекомендации о том, как реализовать функции или методы.

Этот раздел содержит описание набора паттернов безопасности, примеры реализации которых содержатся в составе KasperskyOS Community Edition.

> Паттернам безопасности посвящено множество работ в области информационной безопасности. Для каждого паттерна приводится список работ, использованных при подготовке его описания.

Паттерн Distrustful Decomposition

Описание

При использовании монолитного приложения появляется необходимость дать все необходимые для его работы привилегии одному процессу. Эту проблему решает паттерн `Distrustful Decomposition`.

Целью паттерна `Distrustful Decomposition` является разделение функциональности программы по отдельным процессам, требующим различного уровня привилегий, и контроля взаимодействия между этими процессами вместо создания монолитной программы.

Использование паттерна `Distrustful Decomposition` уменьшает:

- поверхность атаки для каждого из процессов;
- функциональность и данные, которые станут доступны злоумышленнику, если один из процессов будет скомпрометирован.

Альтернативные названия

`Privilege Reduction`.

Контекст

Различные функции приложения требуют разного уровня привилегий.

Проблема

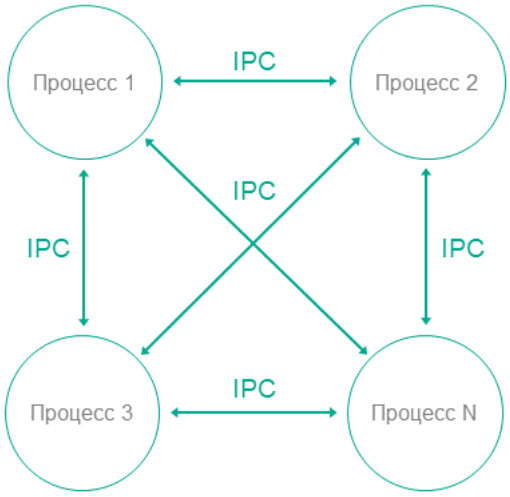
Наивная реализация приложения объединяет множество разнородных по необходимым привилегиям функций в одном компоненте, вынуждая его запускаться с максимальным из необходимых уровней привилегий.

Решение

Паттерн `Distrustful Decomposition` разделяет функциональность по отдельным процессам и изолирует возможные уязвимости в небольшом подмножестве системы. Злоумышленник в случае успешной атаки будет иметь в своем распоряжении функциональность и данные только одного скомпрометированного компонента, но не всего приложения.

Структура

Этот паттерн разбивает одно монолитное приложение на несколько, которые выполняются как отдельные процессы, потенциально имеющие разные привилегии. Каждый процесс реализует небольшой, четко определенный набор функций приложения. Процессы обмениваются данными, используя механизм межпроцессного взаимодействия.



Работа

- В KasperskyOS приложение разбивается на процессы.
- Процессы могут обмениваться сообщениями по IPC.
- Пользователь или удаленная система подключается к процессу, который обеспечивает необходимую функциональность, с уровнем привилегий, достаточным для выполнения запрошенных функций.

Рекомендации по реализации

Взаимодействие между процессами может быть однонаправленным или двунаправленным. Рекомендуется всегда, когда это возможно, использовать однонаправленное взаимодействие, в противном случае увеличивается поверхность атаки на отдельные компоненты и, соответственно, снижается уровень защищенности системы в целом. В случае двустороннего IPC процессы не должны доверять двустороннему обмену данными. Например, если для IPC используется файловая система, то нельзя доверять содержимому файла.

Особенности реализации в KasperskyOS

В универсальных ОС (например Linux, Windows) этот паттерн не использует ничего, кроме стандартной модели процессов/привилегий, уже существующей в этих ОС. Каждая программа выполняется в собственном пространстве процессов с потенциально разными привилегиями пользователя в каждом процессе, однако атака на ядро ОС снижает ценность применения этого паттерна.

Специфика применения этого паттерна при разработке под KasperskyOS состоит в том, что контроль над процессами и IPC возложен на микроядро, атака на которое сложна. Для контроля IPC используется модуль безопасности Kaspersky Security Module.

За счет использования механизмов KasperskyOS достигается высокий уровень надежности программной системы при том же или меньшем объеме усилий разработчика в сравнении с использованием этого же паттерна в программах под универсальные ОС.

Кроме этого, KasperskyOS предоставляет возможность гибкой настройки политик безопасности. При этом процесс задания и изменения политик безопасности потенциально независим от процесса разработки самих приложений.

Связанные паттерны

Использование паттерна `Distrustful Decomposition` предполагает использование паттернов Defer to Kernel и Policy Decision Point.

Примеры реализации

Примеры реализации паттерна `Distrustful Decomposition`:

- Secure Logger
- Separate Storage

Источники

Паттерн `Distrustful Decomposition` подробно рассмотрен в следующих работах:

- 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

Пример Secure Logger

Пример `Secure Logger` демонстрирует использование паттерна Distrustful Decomposition для решения задачи разделения функциональности чтения и записи в журнал событий.

Архитектура примера

Цель безопасности в примере `Secure Logger` заключается в том, чтобы предотвратить возможность искажения или удаления информации в журнале событий. В примере для достижения этой цели безопасности используются возможности, предоставляемые KasperskyOS.

При рассмотрении системы журналирования можно выделить следующие функциональные шаги:

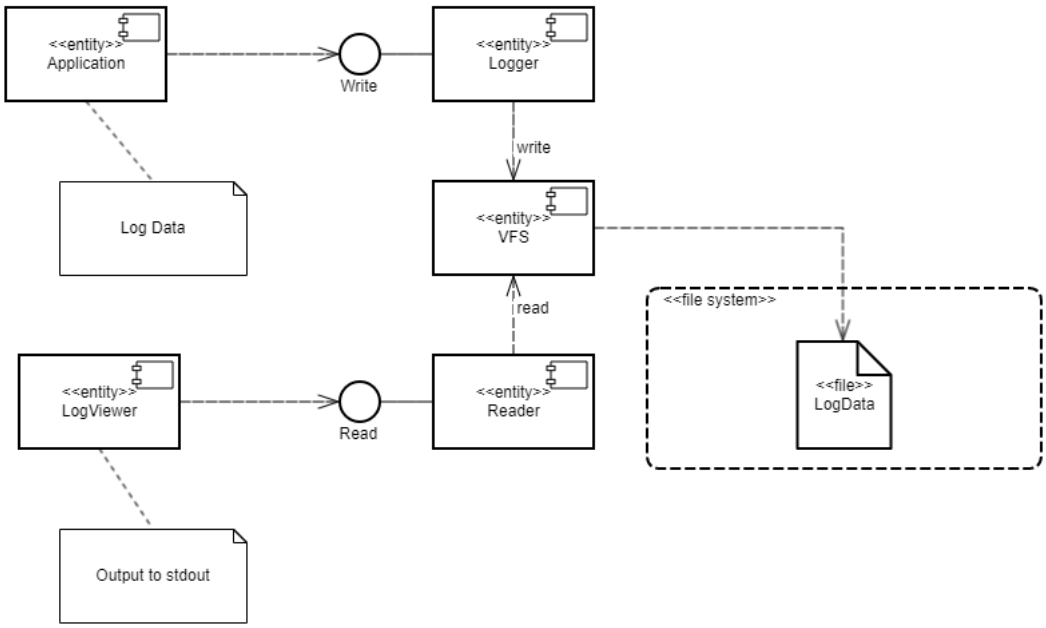
- генерация информации для записи в журнал;
- сохранение информации в журнал;
- чтение записей из журнала;
- предоставление записей в удобном для потребителя виде.

Таким образом, подсистему журналирования можно разделить на четыре процесса в зависимости от необходимых функциональных возможностей каждого процесса.

Для этого пример `Secure Logger` содержит четыре программы: `Application`, `Logger`, `Reader` и `LogViewer`.

- Программа `Application` инициирует создание записей в журнале событий, поддерживаемом программой `Logger`.
- Программа `Logger` создает записи в журнале и записывает их на диск.
- Программа `Reader` читает записи с диска для передачи программе `LogViewer`.
- Программа `LogViewer` передает записи пользователю.

IPC-интерфейс, который предоставляет программа `Logger`, предназначен *только* для записи в хранилище. IPC-интерфейс программы `Reader` предназначен только для чтения из хранилища. Архитектура примера выглядит следующим образом:



- Программа `Application` использует интерфейс программы `Logger` для сохранения записей.
- Программа `LogViewer` использует интерфейс программы `Reader` для чтения записей и предоставления их пользователю.

В общем случае программа `LogViewer` имеет внешние каналы взаимодействия с пользователем (прием команд на чтение данных, предоставление данных пользователю). Очевидно, что эта программа является недоверенным компонентом системы, через которую может проводиться атака. Однако даже в случае успешной атаки, вплоть до внедрения произвольно исполняемого кода в программу `LogViewer`, информация в журнале не будет искажена, так как эта программа может пользоваться только интерфейсом чтения данных, через который искажение или удаление невозможно. При этом `LogViewer` не имеет возможности получить доступ к другим интерфейсам, так как доступ контролируется модулем безопасности.

Политика безопасности в примере `Secure Logger` имеет следующие особенности:

- Программа `Application` имеет возможность обращаться к программе `Logger` для создания новой записи в журнале событий.
- Программа `LogViewer` имеет возможность обращаться к программе `Reader` для чтения записей из журнала событий.
- Программа `Application` *не* имеет возможности обращаться к программе `Reader` для чтения записей из журнала событий.
- Программа `LogViewer` *не* имеет возможности обращаться к программе `Logger` для создания новой записи в журнале событий.

Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

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

Сборка и запуск примера

См. "Сборка и запуск примеров".

Пример Separate Storage

Пример `Separate Storage` демонстрирует использование паттерна Distrustful Decomposition для решения задачи раздельного хранения данных для доверенных и недоверенных приложений.

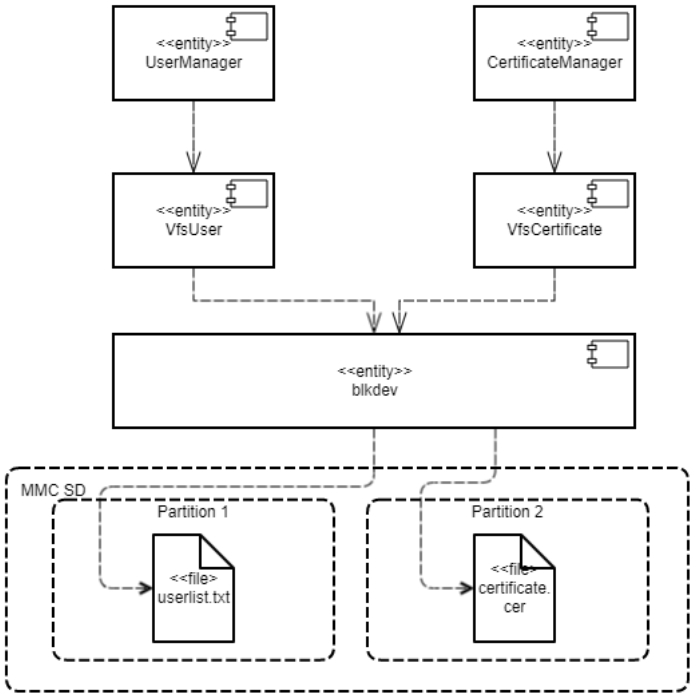
Архитектура примера

Пример `Separate Storage` содержит две пользовательские программы: `UserManager` и `CertificateManager`.

Эти программы работают с данными, которые размещаются в соответствующих файлах:

- Программа `UserManager` работает с данными из файла `userlist.txt`;
- Программа `CertificateManager` работает с данными из файла `certificate.cer`.

Каждая из этих программ использует собственный экземпляр программы VFS для доступа к отдельной файловой системе. При этом каждая программа VFS включает в себя драйвер блочного устройства, связанный с отдельным логическим разделом диска. Программа `UserManager` не имеет доступа к файловой системе программы `CertificateManager` и наоборот.



Такая архитектура гарантирует, что в случае атаки или ошибки в любой из программ `UserManager` и `CertificateManager`, эта программа не сможет получить доступ к файлу, который не предназначен для выполнения ее работы.

Политика безопасности в примере `Separate Storage` имеет следующие особенности:

- Программа `UserManager` имеет доступ к файловой системе *только* через программу `VfsUser`.
- Программа `CertificateManager` имеет доступ к файловой системе через *только* через программу `VfsCertificate`.

Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

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

Сборка и запуск примера

Чтобы запустить пример на QEMU, перейдите в директорию с примером, соберите пример и выполните следующие команды:

```
$ cd build/einit
# Перед выполнением следующей команды убедитесь, что путь к
# директории с исполняемым файлом qemu-system-aarch64 сохранен в
# переменной окружения PATH. В случае отсутствия
# добавьте его в переменную PATH.
$ qemu-system-aarch64 -m 2048 -machine vexpress-a15 -nographic -monitor none -sd
sdcard0.img -kernel kos-qemu-image
```

Также см. "Сборка и запуск примеров".

Подготовка SD-карты для запуска на Raspberry Pi 4 B

Для запуска примера `Separate Storage` на Raspberry Pi 4 B необходимы следующие дополнительные действия:

- создать директорию `/lib` на загрузочном разделе SD-карты, если этой директории не существует;
- скопировать в директорию `/lib` на SD-карте содержимое директории `build/hdd/part1/lib`, которая генерируется во время сборки примера;
- SD-карта, помимо загрузочного раздела с образом решения, должна также содержать 2 дополнительных раздела с файловой системой `ext2` или `ext3`;
- первый дополнительный раздел должен содержать файл `userlist.txt` из директории `./resources/files/`;
- второй дополнительный раздел должен содержать файл `certificate.cer` из директории `./resources/files/`.

> Для запуска примера `Separate Storage` на Raspberry Pi 4 B можно использовать SD-карту, подготовленную для запуска примера `vfs_extfs` на Raspberry Pi 4 B, скопировав файлы `userlist.txt` и `certificate.cer` на соответствующие разделы.

Паттерн Defer to Kernel

Описание

Паттерн `Defer to Kernel` предполагает использование преимущества контроля разрешений на уровне ядра ОС.

Целью этого паттерна является четкое отделение функциональности, требующей повышенных привилегий, от функциональности, не требующей повышенных привилегий, с помощью механизмов, доступных на уровне ядра ОС. Использование механизмов ядра позволяет не реализовывать новых средств для арбитража решений безопасности на уровне пользователя.

Альтернативные названия

`Policy Enforcement Point (PEP)`, `Protected System`, `Enclave`.

Контекст

Паттерн `Defer to Kernel` применим, если система имеет следующие характеристики:

- В системе есть процессы без повышенных привилегий, в том числе пользовательские процессы.
- Некоторые функции системы требуют повышенных привилегий, которые необходимо проверять перед предоставлением процессам доступа к данным.
- Необходимо проверять не только привилегии запрашивающего процесса, но и общую допустимость запрошенной операции в контексте работы всей системы и ее общей безопасности.

Проблема

В условиях разделения функциональности по разным процессам с разным уровнем привилегий необходимо проверять привилегии при выполнении запроса от одного процесса к другому. Выполнять такие проверки и выдавать разрешения должен доверенный код, минимально подверженный атакам. Доверенность прикладного кода почти всегда под вопросом как в силу его объема, так и в силу его направленности на реализацию функциональных требований.

Решение

Отделить привилегированную функциональность и данные от непривилегированных на уровне процессов и отдать ядру ОС контроль межпроцессных взаимодействий (IPC) с проверкой прав доступа при запросе функциональности или данных, требующих повышенных привилегий, а также с проверкой общего состояния системы и состояний отдельных процессов в момент запроса.

Структура

Работа

- Функциональность и управление данными с разными привилегиями разделены между процессами.
- Изоляцию процессов обеспечивает ядро ОС.
- `Процесс-1` хочет запросить привилегированную функциональность или данные у `Процесса-2`, используя IPC.
- Ядро контролирует IPC и разрешает или не разрешает коммуникацию исходя из политик безопасности и доступной ему информации о контексте работы и состоянии `Процесса-1`.

Рекомендации по реализации

Для того чтобы конкретная реализация паттерна работала безопасно и надежно, необходимо следующее:

- **Изоляция**

  Необходимо обеспечить полную и гарантированную изоляцию процессов.

- **Невозможность обойти ядро**

  Абсолютно все IPC-взаимодействия должны контролироваться ядром.

- **Самозащита ядра**

  Необходимо обеспечить доверенность ядра, его собственную защиту от компрометации.

- **Доказуемость**

  Требуется определенный уровень гарантий безопасности и надежности в отношении ядра.

- **Возможность внешнего вычисления разрешений о доступе**

  Необходимо, чтобы разрешения о доступе вычислялись на уровне ОС, а не были реализованы в прикладном коде.

  Для этого, в частности, необходимо предоставить инструменты для описания политик доступа, чтобы политики безопасности были отделены от бизнес-логики.

Особенности реализации в KasperskyOS

Ядро KasperskyOS гарантирует изоляцию процессов и представляет собой Policy Enforcement Point (PEP).

Связанные паттерны

Паттерн `Defer to Kernel` является частным случаем паттернов Distrustful Decomposition и Policy Decision Point. Паттерн `Policy Decision Point` определяет абстрактный процесс, перехватывающий все запросы к ресурсам и проверяющий их на соответствие заданной политике безопасности. Специфика паттерна `Defer to Kernel` в том, что эту проверку выполняет ядро ОС – это более надежное и портируемое решение, сокращающее время разработки и тестирования.

Следствия

Перенос ответственности за применение политики доступа на ядро ОС приводит к отделению политики безопасности от бизнес-логики (которая может быть очень сложна), что упрощает разработку и повышает портируемость за счет использования функций ядра ОС.

Кроме этого, появляется возможность доказать безопасность решения в целом, доказав правильность работы ядра. Сложность доказуемости правильной работы кода нелинейно растет с увеличением его размера. Паттерн `Defer to Kernel` минимизирует объем доверенного кода – при условии, что ядро ОС невелико.

Примеры реализации

Пример реализации паттерна `Defer to Kernel`: Пример Defer to Kernel.

Источники

Паттерн `Defer to Kernel` подробно рассмотрен в следующих работах:

- 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).

Пример Defer to Kernel

Пример `Defer to Kernel` демонстрирует использование паттернов Defer to Kernel и Policy Decision Point.

Пример `Defer to Kernel` содержит три пользовательские программы: `PictureManager`, `ValidPictureClient` и `NonValidPictureClient`.

В этом примере программы `ValidPictureClient` и `NonValidPictureClient` обращаются к программе `PictureManager` для получения информации.

Только программе `ValidPictureClient` разрешено взаимодействие с программой `PictureManager`.

Ядро KasperskyOS гарантирует изоляцию запущенных программ (процессов).

Контроль взаимодействия программ в KasperskyOS вынесен в модуль безопасности Kaspersky Security Module. Эта подсистема анализирует каждый отправляемый запрос и ответ и на основе заданной политики безопасности выносит решение: разрешить или запретить его доставку.

Политика безопасности в примере `Defer to Kernel` имеет следующие особенности:

- Программе `ValidPictureClient` явно разрешено взаимодействие с программой `PictureManager`.
- Программе `NonValidPictureClient` взаимодействие с программой `PictureManager` *не* разрешено явно. Таким образом, это взаимодействие запрещено (*принцип Default Deny*).

Динамическое создание IPC-каналов

Пример также демонстрирует возможность динамического создания IPC-каналов между процессами. Динамическое создание IPC-каналов осуществляется с помощью сервера имен – специального сервиса ядра, представленного программой `NameServer`. Возможность динамического создания IPC-каналов позволяет изменять топологию взаимодействия программ "на лету".

Любая программа, которой разрешено взаимодействие с `NameServer` по IPC, может зарегистрировать в сервере имен свои интерфейсы. Другая программа может запросить у сервера имен зарегистрированные интерфейсы, после чего осуществить подключение к нужному интерфейсу.

При этом все взаимодействия по IPC (даже созданные динамически) контролируются с помощью модуля безопасности.

Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

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

Сборка и запуск примера

См. "Сборка и запуск примеров".

Паттерн Policy Decision Point

Описание

Паттерн `Policy Decision Point` предполагает инкапсуляцию вычисления решений на основе методов моделей безопасности в отдельный компонент системы, который обеспечивает выполнение этих методов безопасности в полном объеме и в правильной последовательности.

Альтернативные названия

`Check Point`, `Access Decision Function`.

Контекст

Система имеет функции с разным уровнем привилегий, а политика безопасности нетривиальна (содержит много привязок методов моделей безопасности к событиям безопасности).

Проблема

Если проверки соблюдения политики безопасности разнесены по разным компонентам системы, возникают следующие проблемы:

- необходимо тщательно контролировать, что выполняются все необходимые проверки во всех необходимых случаях;
- сложно обеспечивать правильный порядок выполнения проверок;
- сложно доказать правильность работы системы проверок, ее целостность и непротиворечивость;
- политика безопасности связана с бизнес-логикой, поэтому ее изменение влечет необходимость менять бизнес-логику, что усложняет поддержку и увеличивает вероятность ошибок.

Решение

Все проверки соблюдения политики безопасности проводятся в отдельном компоненте Policy Decision Point (PDP). Этот компонент отвечает за обеспечение правильного порядка проверок и за их полноту. Происходит отделение проверки политики от кода, реализующего бизнес-логику.

Структура

Работа

- Policy Enforcement Point (PEP) получает запрос на доступ к функциональности или данным.

  PEP может представлять собой, например, ядро ОС. Подробнее см. Паттерн Defer to Kernel.

- PEP собирает атрибуты запроса, необходимые для принятия решений по управлению доступом.

- PEP запрашивает решение по управлению доступом у Policy Decision Point (PDP).
- PDP вычисляет решение о предоставлении доступа на основе политики безопасности и информации, полученной в запросе от PEP.
- PEP отклоняет или разрешает взаимодействие на основе решения PDP.

Рекомендации по реализации

При реализации необходимо учитывать проблему "Время проверки vs. Время использования". Например, если политика безопасности зависит от быстро меняющегося статуса какого-либо объекта системы, вычисленное решение так же быстро теряет актуальность. В системе, использующей паттерн **Policy Decision Point**, необходимо позаботиться о минимизации интервала между принятием решения о доступе и моментом выполнения запроса на основе этого решения.

Особенности реализации в KasperskyOS

Ядро KasperskyOS гарантирует изоляцию процессов и представляет собой Policy Enforcement Point (PEP).

Контроль взаимодействия процессов в KasperskyOS вынесен в модуль безопасности Kaspersky Security Module. Этот модуль анализирует каждый отправляемый запрос и ответ и на основе заданной политики безопасности выносит решение: разрешить или запретить его доставку. Таким образом, Kaspersky Security Module выполняет роль Policy Decision Point (PDP).

Следствия

Паттерн позволяет настраивать политику безопасности без внесения изменений в код, реализующий бизнес-логику, и делегировать сопровождение системы с точки зрения информационной безопасности.

Связанные паттерны

Использование паттерна `Policy Decision Point` предполагает использование паттернов Distrustful decomposition и Defer to Kernel.

Примеры реализации

Пример реализации паттерна `Policy Decision Point`: Пример Defer to Kernel.

Источники

Паттерн `Policy Decision Point` подробно рассмотрен в следующих работах:

- 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

Паттерн Privilege Separation

Описание

Паттерн `Privilege Separation` предполагает использование непривилегированных изолированных модулей системы для взаимодействия с клиентами (другими модулями или пользователями), которые не имеют привилегий. Целью паттерна `Privilege Separation` является уменьшение количества кода, выполняемого с особыми привилегиями, не влияющее на функциональность программы и не ограничивающее ее.

Паттерн `Privilege Separation` является частным случаем паттерна Distrustful Decomposition.

Пример

Неаутентифицированный пользователь подключается к системе, в которой есть функции, требующие повышенных привилегий.

Контекст

В системе есть компоненты с большой поверхностью атаки из-за большого числа связей с ненадежными источниками и/или сложной, потенциально подверженной ошибкам реализации.

Проблема

Когда клиент, имеющий неизвестные привилегии, взаимодействует с привилегированным компонентом системы, возникают риски компрометации данных и функциональности, доступных этому компоненту.

Решение

Взаимодействие с ненадежными клиентами необходимо вести только через специально выделенные компоненты, у которых нет привилегий. Важно, что паттерн `Privilege Separation` не изменяет функциональность системы, он лишь разделяет функциональность на компоненты с разными привилегиями.

Работа

Работа паттерна делится на две фазы:

- **Pre-Authentication.** Клиент еще не аутентифицирован. Он отправляет запрос к привилегированному мастер-процессу. Мастер-процесс создает дочерний процесс, лишенный привилегий (в том числе, доступа к файловой системе), который выполняет аутентификацию клиента.

- **Post-Authentication.** Клиент аутентифицирован и авторизован. Привилегированный мастер-процесс создает новый дочерний процесс, обладающий привилегиями, соответствующими правам клиента. Этот процесс отвечает за все дальнейшее взаимодействие с клиентом.

Рекомендации по реализации в KasperskyOS

На этапе **Pre-Authentication** мастер-процесс может хранить состояние каждого непривилегированного процесса в виде конечного автомата и изменять состояние автомата при аутентификации.

Запросы дочерних процессов к мастер-процессу выполняются с использованием стандартных механизмов IPC. При этом контроль взаимодействий осуществляется с помощью модуля безопасности Kaspersky Security Module.

Следствия

Если атакующий получает контроль над непривилегированным процессом, он не получит доступа ни к каким привилегированным функциям или данным. Если он получает контроль над авторизованным процессом, он получит только привилегии этого процесса.

Кроме того, организованный таким образом код проще проверять и тестировать – особого внимания требует лишь функциональность, работающая с повышенными привилегиями.

Примеры реализации

Пример реализации паттерна `Privilege Separation`: Пример Device Access.

Источники

Паттерн `Privilege Separation` подробно рассмотрен в следующих работах:

- 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

Пример Device Access

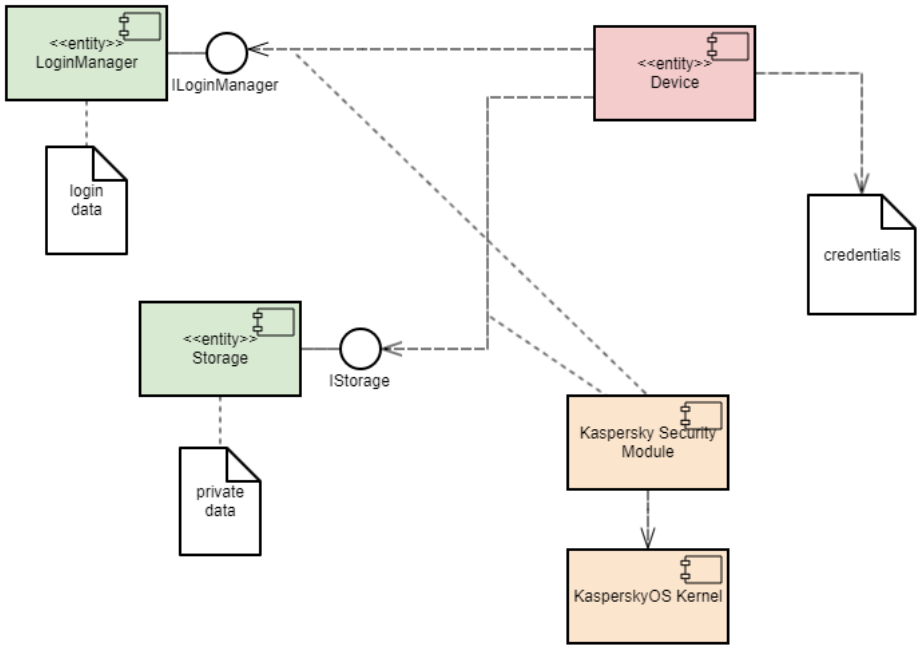
Пример `Device Access` демонстрирует использование паттерна Privilege Separation.

Архитектура примера

Пример содержит три программы: `Device`, `LoginManager` и `Storage`.

В этом примере программа `Device` обращается к программе `Storage` для получения информации и к программе `LoginManager` для авторизации.

Программа `Device` получает доступ к программе `Storage` только после успешной авторизации.



Пример демонстрирует возможность разделения логики авторизации и логики доступа к данным на независимые компоненты. Такое разделение гарантирует, что доступ к данным может быть открыт только после успешной авторизации. При этом контроль за тем, что авторизация была проведена и закончилась успешно, осуществляется модулем безопасности. Кроме этого, такая архитектура позволяет производить независимую разработку и тестирование логики авторизации и логики предоставления доступа к данным.

Политика безопасности в примере `Device Access` имеет следующие особенности:

- Программа `Device` имеет возможность обращаться к программе `LoginManager` для авторизации.
- Вызовами метода `GetInfo()` программы `Storage` управляют методы модели безопасности Flow:
  - Конечный автомат, описанный в конфигурации объекта `session`, имеет два состояния: `unauthenticated` и `authenticated`.
  - Исходное состояние – `unauthenticated`.
  - Разрешены переходы из `unauthenticated` в `authenticated` и обратно.
  - Объект `session` создается при запуске программы `Device`.
  - При успешном вызове программой `Device` метода `Login()` программы `LoginManager` состояние объекта `session` изменяется на `authenticated`.
  - При успешном вызове программой `Device` метода `Logout()` программы `LoginManager` состояние объекта `session` изменяется на `unauthenticated`.
  - При вызове программой `Device` метода `GetInfo()` программы `Storage` проверяется текущее состояние объекта `session`. Вызов разрешается, только если текущее состояние объекта – `authenticated`.

Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

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

Сборка и запуск примера

См. "Сборка и запуск примеров".

Паттерн Information Obscurity

Описание

Цель паттерна `Information Obscurity` – шифрование конфиденциальных данных в небезопасных средах с целью защиты данных от кражи.

Контекст

Этот паттерн следует использовать, когда данные часто передаются между частями системы и/или между системой и другими (внешними) системами.

Проблема

Конфиденциальные данные могут передаваться через недоверенную среду как внутри одной системы (через недоверенные компоненты), так и между разными системами (через недоверенные сети). В случае компрометации этой среды конфиденциальные данные могут быть получены злоумышленником.

Решение

Данные должны быть разделены по уровню конфиденциальности, чтобы определить, какие данные следует зашифровать и какие алгоритмы шифрования использовать. Поскольку шифрование и дешифрование могут занять много времени, лучше по возможности ограничить их использование. Паттерн `Information Obscurity` решает эту проблему за счет использования уровня конфиденциальности для определения того, что необходимо скрыть с помощью шифрования.

Примеры реализации

Пример реализации паттерна `Information Obscurity`: Пример Secure Login.

Источники

Паттерн `Information Obscurity` подробно рассмотрен в следующих работах:

- 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).

Пример Secure Login (Civetweb, TLS-terminator)

Пример `Secure Login` демонстрирует использование паттерна Information Obscurity. Пример демонстрирует возможность передачи критической для системы информации через недоверенную среду.

Архитектура примера

В примере имитируется получение удаленного доступа к IoT-устройству посредством передачи этому устройству учетных данных пользователя (имени пользователя и пароля). Недоверенной средой внутри IoT-устройства является веб-сервер, который обслуживает запросы пользователей. Практика показывает, что такой веб-сервер является легко обнаруживаемым и зачастую успешно атакуемым, так как IoT-устройства не имеют встроенных средств защиты от проникновения и других атак. Кроме того, пользователи получают доступ к IoT-устройству через недоверенную сеть. Очевидно, что в таких условиях для защиты учетных данных пользователя от компрометации необходимо использовать криптографические алгоритмы.

С точки зрения архитектуры в таких системах можно выделить следующие субъекты:

- Источник данных: браузер пользователя.
- Точка коммуникации с устройством: веб-сервер.
- Подсистема обработки информации от пользователя: подсистема аутентификации.

При этом для использования криптографической защиты необходимо выполнить следующие шаги:

1. Обеспечить взаимодействие источника данных и устройства по протоколу HTTPS. Это позволит избежать "прослушивания" HTTP-трафика и атак типа MITM (man in the middle).
2. Выработать между источником данных и подсистемой обработки информации общий секрет.
3. Использовать этот секрет для шифрования информации на стороне источника данных и расшифровки на стороне подсистемы обработки информации. Это позволит избежать компрометации данных внутри устройства (в точке коммуникации).

Пример `Secure Login` включает следующие компоненты:

- Веб-сервер `Civetweb` (недоверенный компонент, программа `WebServer`).
- Подсистему аутентификацию пользователей (доверенный компонент, программа `AuthService`).
- TLS-терминатор (доверенный компонент, программа `TlsEntity`). Этот компонент поддерживает транспортный механизм TLS (transport layer security). TLS-терминатор совместно с веб-сервером поддерживают протокол HTTPS на стороне устройства (веб-сервер взаимодействует с браузером через TLS-терминатор).

Процесс аутентификации пользователя происходит по следующей схеме:

1. Пользователь открывает в браузере страницу по адресу `https://localhost:1106` (при запуске примера на QEMU) или по адресу `https://<IP-адрес Raspberry Pi>:1106` (при запуске примера на Raspberry Pi 4 B). HTTP-трафик между браузером и TLS-терминатором будет передаваться в зашифрованном виде, а веб-сервер будет работать с открытым HTTP-трафиком.

> В примере используется самоподписанный сертификат, поэтому большинство современных браузеров сообщит о незащищенности соединения. Нужно согласиться использовать незащищенное соединение, которое тем не менее будет зашифрованным. В некоторых браузерах возможно возникновение сообщения `"TLS: Error performing handshake: -30592: errno = Success"`.

2. Веб-сервер `Civetweb`, запущенный в программе `WebServer`, отображает страницу `index.html`, содержащую приглашение к аутентификации.

3. Пользователь нажимает на кнопку `Log in`.

4. Программа `WebServer` обращается к программе `AuthService` по IPC для получения страницы, содержащей форму ввода имени пользователя и пароля.

5. Программа `AuthService` выполняет следующие действия:

   - генерирует закрытый ключ, открытые параметры, а также вычисляет открытый ключ по алгоритму Диффи-Хеллмана;
   - создает страницу `auth.html` с формой ввода имени пользователя и пароля (код страницы содержит открытые параметры и открытый ключ);
   - передает полученную страницу программе `WebServer` по IPC.

6. Веб-сервер `Civetweb`, запущенный в программе `WebServer`, отображает страницу `auth.html` с формой ввода имени пользователя и пароля.

7. Пользователь заполняет форму и нажимает на кнопку `Submit` (корректные данные для аутентификации содержатся в файле `secure_login/auth_service/src/authservice.cpp`).

8. Код страницы `auth.html`, который исполняется на стороне браузера, осуществляет следующие действия:

   - генерирует закрытый ключ, вычисляет открытый ключ и общий секретный ключ по алгоритму Диффи-Хеллмана;
   - выполняет шифрование пароля операцией `XOR` с использование общего секретного ключа;
   - передает веб-северу имя пользователя, зашифрованный пароль и открытый ключ.

9. Программа `WebServer` обращается к программе `AuthService` по IPC для получения страницы, содержащей результат аутентификации, передавая имя пользователя, зашифрованный пароль и открытый ключ.

10. Программа `AuthService` выполняет следующие действия:

    - вычисляет общий секретный ключ по алгоритму Диффи-Хеллмана;
    - расшифровывает пароль с использованием общего секретного ключа;
    - возвращает страницу `result_err.html` или страницу `result_ok.html` в зависимости от результата аутентификации.

11. Веб-сервер `Civetweb`, запущенный в программе `WebServer`, отображает страницу `result_err.html` или страницу `result_ok.html`.

Таким образом, конфиденциальные данные передаются через сеть и веб-сервер только в зашифрованном виде. Кроме того, весь HTTP-трафик передается через сеть в зашифрованном виде. Для передачи данных между компонентами используются взаимодействия по IPC, которые контролируются модулем Kaspersky Security Module.

Unit-тестирование с использованием фреймворка GoogleTest

Помимо паттерна Information Obscurity пример `Secure Login` демонстрирует использование фреймворка GoogleTest для выполнения unit-тестирования программ, разработанных под KasperskyOS (KasperskyOS Community Edition содержит в своем составе этот фреймворк).

Исходный код тестов находится по следующему пути:

`/opt/KasperskyOS-Community-Edition-<version>/examples/secure_login/tests`

Эти unit-тесты предназначены для верификации некоторых cpp-модулей подсистемы аутентификации и веб-сервера.

Чтобы запустить тестирование, выполните следующие действия:

1. Перейдите в директорию с примером `Secure Login`.
2. Удалите директорию `build` с результатами предыдущей сборки, выполнив команду:

   ```
   sudo rm -rf build/
   ```

3. Выполните команду для запуска тестирования:

   ```
   $ RUN_TESTS=YES ./cross-build.sh
   ```

Тесты выполняются в программе `TestEntity`. Программы `AuthService` и `WebServer` не запускаются, поэтому при выполнении тестирования пример нельзя использовать для демонстрации паттерна Information Obscurity.

После завершения тестирования выводятся результаты выполнения тестов.

Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

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

Сборка и запуск примера

Чтобы запустить пример на QEMU, перейдите в директорию с примером, соберите пример и выполните следующие команды:

```
$ cd build/einit
# Перед выполнением следующей команды убедитесь, что путь к
# директории с исполняемым файлом qemu-system-aarch64 сохранен в
# переменной окружения PATH. В случае отсутствия
# добавьте его в переменную PATH.
$ qemu-system-aarch64 -m 2048 -machine vexpress-a15 -nographic -monitor none -net
nic,macaddr=52:54:00:12:34:56 -net user,hostfwd=tcp::1106-:1106 -sd sdcard0.img -
kernel kos-qemu-image
```

Также см. "Сборка и запуск примеров".

Для корректной работы примера `secure_login` на Raspberry Pi после сборки примера и подготовки загрузочной SD-карты требуется выполнить следующие действия:

- скопировать директории `certs` и `www`, расположенные по пути `/opt/KasperskyOS-Community-Edition-<version>/examples/secure_login/resources/hdd`, в корневую директорию загрузочной SD-карты;
- создать директорию `/lib` на загрузочной SD-карте, если этой директории не существует;
- скопировать в директорию `/lib` на загрузочной SD-карте содержимое директории `build/hdd/lib`, которая генерируется во время сборки примера.

Приложения

Этот раздел содержит информацию, которая дополняет основной текст документа.

Дополнительные примеры

Этот раздел содержит описания дополнительных примеров, входящих в состав KasperskyOS Community Edition.

См. также описания примеров реализации паттернов безопасности:

- Пример Secure Logger
- Пример Separate Storage
- Пример Defer to Kernel
- Пример Device Access
- Пример Secure login (Civetweb, TLS-terminator)

Пример hello

Код `hello.c` выглядит привычным и простым для разработчика на языке C – он полностью совместим с POSIX:

hello.c

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, const char *argv[])

{

    fprintf(stderr,"Hello world!\n");

    return EXIT_SUCCESS;

}
```

Скомпилируйте этот код с использованием `aarch64-kos-gcc` (входит в состав средств разработки KasperskyOS Community Edition):

```
aarch64-kos-gcc -o Hello hello.c
```

> Имя программы (а значит и имя исполняемого файла) должно начинаться с заглавной буквы.

EDL-описание класса процессов Hello

Статическое описание программы `Hello` состоит из единственного файла `Hello.edl`, в котором необходимо прописать имя класса процессов:

Hello.edl

```
/* После ключевого слова "entity" указано имя класса процессов. */
entity Hello
```

> Имя класса процессов должно начинаться с заглавной буквы. Имя EDL-файла должно совпадать с именем класса, который он описывает.

Создание инициализирующей программы Einit

> При загрузке KasperskyOS ядро запускает программу с именем `Einit`. Программа `Einit` запускает все остальные программы, входящие в решение, то есть служит *инициализирующей программой*.
> В составе пакета инструментов KasperskyOS Community Edition поставляется утилита einit, которая позволяет сгенерировать код инициализирующей программы (`einit.c`) на основе *init-описания*. В приведенном ниже примере файл с init-описанием называется `init.yaml`, хотя может иметь любое имя. Подробнее см. "Запуск процессов".

Для того чтобы программа `Hello` запустилась после загрузки операционной системы, достаточно указать ее имя в файле `init.yaml` и собрать на его основе программу `Einit`.

init.yaml

```
entities:
# Запустить программу "Hello".
- name: Hello
```

Сборка модуля безопасности

Пример hello содержит простейшую политику безопасности решения (`security.psl`), разрешающую любые взаимодействия.

Модуль безопасности (`ksm.module`) собирается на основе `security.psl`.

Файлы примера

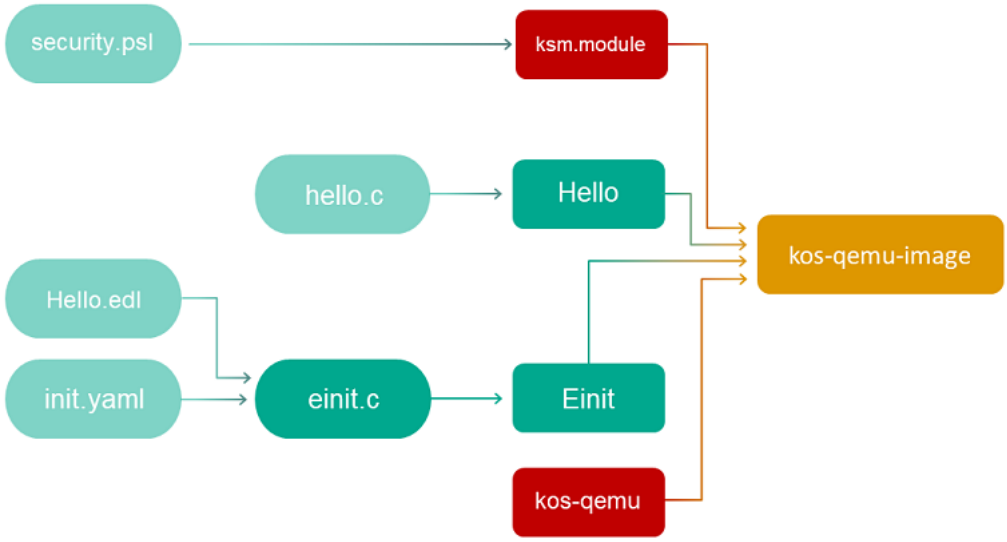
Код примера и скрипты для сборки находятся по следующему пути:

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

Сборка и запуск примера

См. "Сборка и запуск примеров".

Общая схема сборки примера hello выглядит следующим образом:



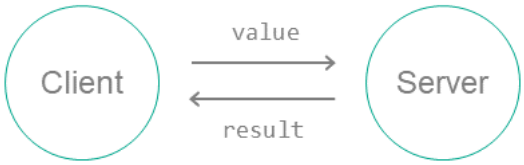
Пример echo

Пример echo демонстрирует использование IPC-транспорта.

Показана работа с основными инструментами, позволяющими реализовать взаимодействие между программами.

Пример echo описывает простейший случай взаимодействия двух программ:

1. Программа `Client` передает программе `Server` число (`value`).
2. Программа `Server` изменяет это число и передает новое число (`result`) программе `Client`.
3. Программа `Client` выводит число `result` на экран.

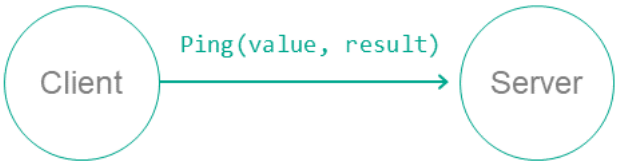


Чтобы организовать такое взаимодействие программ, потребуется:

1. Соединить программы `Client` и `Server`, используя init-описание.
2. Реализовать на сервере интерфейс с единственным методом `Ping`, который имеет один входной аргумент – исходное число (`value`) и один выходной аргумент – измененное число (`result`).

   Описание метода `Ping` на языке IDL:

```
Ping(in UInt32 value, out UInt32 result);
```



3. Создать файлы статических описаний на языках EDL, CDL и IDL. С помощью компилятора NK сгенерировать файлы, содержащие транспортные методы и типы (прокси-объект, диспетчеры и т.д.).

4. В коде программы `Client` инициализировать все необходимые объекты (транспорт, прокси-объект, структуру запроса и др.) и вызвать интерфейсный метод.

5. В коде программы `Server` подготовить все необходимые объекты (транспорт, диспетчер компонента и диспетчер программы и др.), принять запрос от клиента, обработать его и отправить ответ.

Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

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

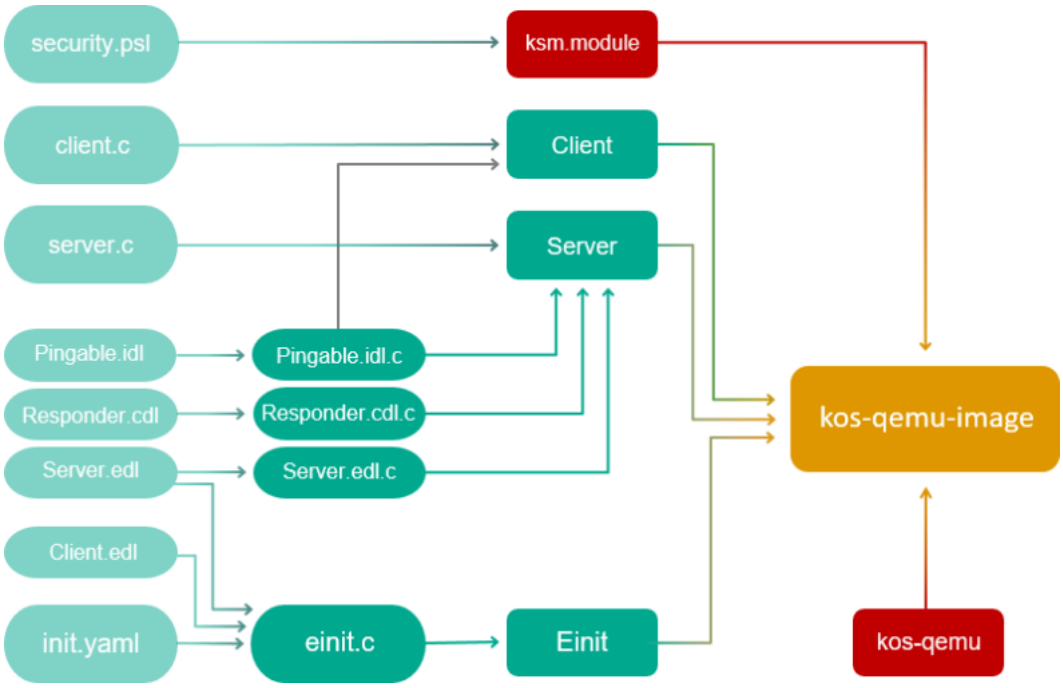
Пример echo состоит из следующих исходных файлов:

- `client/src/client.c` – реализация программы `Client`;
- `server/src/server.c` – реализация программы `Server`;
- `resources/Server.edl`, `resources/Client.edl`, `resources/Responder.cdl`, `resources/Pingable.idl` – статические описания;
- `init.yaml` – init-описание.

Сборка и запуск примера

См. "Сборка и запуск примеров".

Схема сборки примера echo выглядит следующим образом:



Пример ping

Пример ping демонстрирует использование политики безопасности решения для управления взаимодействиями между программами.

Пример ping включает в себя четыре программы: `Client`, `Server`, `KlogEntity` и `KlogStorageEntity`.

Программа `Server` предоставляет два идентичных метода `Ping` и `Pong`, которые получают число и возвращают измененное число:

```
Ping(in UInt32 value, out UInt32 result);
Pong(in UInt32 value, out UInt32 result);
```

Программа `Client` вызывает оба этих метода в различной последовательности. Если вызов метода запрещен политикой безопасности решения, выводится сообщение о неудачной попытке вызова.

Системные программы `KlogEntity`, `KlogStorageEntity` выполняют аудит безопасности.

> Транспортная часть примера ping практически аналогична таковой для примера echo. Единственное отличие состоит в том, что в примере ping используется два метода (`Ping` и `Pong`), а не один.

Политика безопасности решения в примере ping

Политика безопасности решения в этом примере разрешает запуск ядра KasperskyOS и программы Einit, которой разрешено запускать все программы в решении. Обращениями к программе `Server` управляют методы модели безопасности Flow.

Конечный автомат, описанный в конфигурации объекта `request_state` модели безопасности Flow, имеет два состояния: `not_sent` и `sent`. Исходное состояние – `not_sent`. Разрешены только переходы из `not_sent` в `sent` и обратно.

При вызове методов `Ping` и `Pong` проверяется текущее состояние объекта `request_state`. В состоянии `not_sent` разрешен только вызов `Ping`, при этом состояние изменится на `sent`. Аналогично, в состоянии `sent` разрешен только вызов `Pong`, при этом состояние изменится на `not_sent`.

Таким образом, методы `Ping` и `Pong` разрешено вызывать только по очереди.

Фрагмент файла security.psl

```
/* Политика безопасности решения для демонстрации использования модели
 * безопасности Flow в примере ping */

/* Включение PSL-файлов с формальными представлениями моделей безопасности
 * Base и Flow */
use nk.base._
use nk.flow._

/* Включение EDL-файлов */
use EDL Einit
use EDL ping.Client
use EDL ping.Server

/* Создание объекта модели безопасности Flow */
policy object request_state : Flow {
    type States = "not_sent" | "sent"
    config = {
        states      : [ "not_sent", "sent" ],
        initial     : "not_sent",
        transitions : {
            "not_sent" : [ "sent" ],
            "sent"     : [ "not_sent" ]
        }
    }
}

/* При запуске программой Einit программы Server
 * устанавливается начальное состояние конечного автомата */
execute src=Einit dst=ping.Server method=main {
    request_state.init { sid: dst_sid }
}

/* При вызове клиентом класса ping.Client метода Ping службы
controlimpl.connectionimpl
 * сервера класса ping.Server проверяется, что объект request_state находится
 * в состоянии "not_sent". Если это так, то получение запроса разрешается и
 * объект request_state устанавливается в состояние "sent". */
request src=ping.Client dst=ping.Server endpoint=controlimpl.connectionimpl
method=Ping {
    request_state.allow { sid: dst_sid, states: [ "not_sent" ] }
    request_state.enter { sid: dst_sid, state: "sent" }
}

/* При вызове клиентом класса ping.Client метода Pong службы
controlimpl.connectionimpl
 * сервера класса ping.Server проверяется, что объект request_state находится
 * в состоянии "sent". Если это так, то получение запроса разрешается и
 * объект request_state устанавливается в состояние "not_sent". */
request src=ping.Client dst=ping.Server endpoint=controlimpl.connectionimpl
method=Pong {
    request_state.allow { sid: dst_sid, states: [ "sent" ] }
    request_state.enter { sid: dst_sid, state: "not_sent" }
}
```

```
/* Серверу класса ping.Server разрешено отвечать на обращения клиента класса
ping.Client,
 * который вызывает методы Ping и Pong службы controlimpl.connectionimpl. */
response src=ping.Server dst=ping.Client endpoint=controlimpl.connectionimpl {
    match method=Ping { grant () }
    match method=Pong { grant () }
}
```

Описание политики безопасности в примере ping также содержит секцию тестов политики безопасности решения.

Пример такой политики см. в секции "Пример 2" раздела "Примеры тестов политик безопасности решений на базе KasperskyOS".

Полное описание политики безопасности примера ping находится в файлах `security.psl.in` и `core.psl` по следующему пути: `/opt/KasperskyOS-Community-Edition-<version>/examples/ping/einit/src`.

Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

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

Сборка и запуск примера

См. "Сборка и запуск примеров".

Пример net_with_separate_vfs

Пример представляет собой простейший случай взаимодействия по сети с использованием сокетов Беркли.

Пример состоит из программ `Client` и `Server`, связанных TCP-сокетом с использованием loopback-интерфейса. В коде программ используются стандартные POSIX-функции.

Чтобы соединить программы сокетом через loopback, они должны использовать один экземпляр сетевого стека, то есть взаимодействовать с "общей" программой VFS (в этом примере программа называется `NetVfs`).

Для сборки и запуска примера используется система CMake из состава KasperskyOS Community Edition.

Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

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

Сборка и запуск примера

См. "Сборка и запуск примеров".

Пример net2_with_separate_vfs

Пример демонстрирует особенности решения, в котором программа использует стандартные функции POSIX для взаимодействия с внешним сервером.

Пример `net2_with_separate_vfs` является видоизмененным примером `net_with_separate_vfs`. В отличие от примера `net_with_separate_vfs`, в этом примере программа взаимодействует по сети не с другой программой, запущенной в KasperskyOS, а с внешним сервером.

Пример состоит из программы `Client`, запущенной в KasperskyOS под QEMU или на Raspberry Pi, и программы `Server`, запущенной в хостовой операционной системе Linux. Программа `Client` и программа `Server` связаны TCP-сокетом. В коде программы `Client` используются стандартные функции POSIX.

Чтобы соединить программы `Client` и `Server` сокетом, программа `Client` должна взаимодействовать с программой `NetVfs`. Программа `NetVfs` при сборке компонуется с сетевым драйвером, который обеспечит взаимодействие с программой `Server`, запущенной в Linux.

Для сборки и запуска примера используется система CMake из состава KasperskyOS Community Edition.

Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

```
/opt/KasperskyOS-Community-Edition-<version>/examples/net2_with_separate_vfs
```

Сборка и запуск примера

См. "Сборка и запуск примеров".

Для корректной работы примера необходимо запустить программу `Server` в хостовой операционной системе Linux или на компьютере, подключенном к Raspberry Pi.

После выполнения сборки, исполняемый файл `server` программы `Server` находится в следующей директории:

```
/opt/KasperskyOS-Community-Edition-
<version>/examples/net2_with_separate_vfs/build/host/server/
```

Чтобы собрать исполняемый файл программы `Server` самостоятельно, нужно выполнить следующие команды:

```
$ cd net2_with_separate_vfs/server/src/
$ gcc -o server server.c
```

Пример embedded_vfs

Пример показывает, как встроить виртуальную файловую систему (далее VFS), поставляемую в составе KasperskyOS Community Edition, в разрабатываемую программу.

В этом примере программа `Client` полностью инкапсулирует реализацию VFS из KasperskyOS Community Edition. Это позволяет избавиться от использования IPC для всех стандартных функций ввода-вывода (`stdio.h`, `socket.h` и так далее), например, для отладки или повышения производительности.

Программа `Client` тестирует следующие операции:

- создание директории;
- создание и удаление файла;
- чтение из файла и запись в файл.

Поставляемые ресурсы

В пример входит образ жесткого диска с файловой системой FAT32 – `hdd.img`.

Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

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

Сборка и запуск примера

См. "Сборка и запуск примеров".

Пример vfs_extfs

Пример показывает, как встроить новую файловую систему в виртуальную файловую систему (VFS), поставляемую в составе KasperskyOS Community Edition.

В этом примере программа `Client` тестирует работу файловых систем (`ext2`, `ext3`, `ext4`) на блочных устройствах. Для этого `Client` обращается по IPC к виртуальной файловой системе (программе `FileVfs`), а `FileVfs` в свою очередь обращается по IPC к блочному устройству.

Файловые системы `ext2` и `ext3` работают с настройками по умолчанию. Файловая система `ext4` работает, если отключить `extent` (`mkfs.ext4 -O ^64bit,^extent /dev/foo`).

Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

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

Сборка и запуск примера

См. "Сборка и запуск примеров".

Подготовка SD-карты для запуска на Raspberry Pi 4 B

Для запуска примера `vfs_extfs` на Raspberry Pi 4 B необходимо, чтобы SD-карта, помимо загрузочного раздела с образом решения, также содержала 3 дополнительных раздела с файловыми системами `ext2`, `ext3` и `ext4` соответственно.

Пример multi_vfs_ntpd

Этот пример показывает как использовать ntp-сервис в KasperskyOS. Программа `Ntpd` поставляется в составе KasperskyOS Community Edition и представляет собой реализацию ntp-клиента, который в фоновом режиме получает параметры времени от внешних ntp-серверов и передает их ядру KasperskyOS.

Пример также демонстрирует использование разных виртуальных файловых систем (далее VFS) в одном решении:

- для работы с сетью используется программа `VfsNet`;
- для работы с файловой системой используется программа `VfsSdCardFs`.

Программа `Client` использует стандартные функции библиотеки `libc` для получения информации о времени, которые транслируются в обращения к программе VFS по IPC.

Для сборки и запуска примера используется система CMake из состава KasperskyOS Community Edition.

Поставляемые ресурсы

- В директории `./resources/edl` расположен файл `Client.edl`, который содержит статическое описание программы `Client`.
- В директории `./resources/hdd/etc` расположены файлы конфигурации для программ `VfsNet`, `Dhcpcd` и `Ntpd`: `hosts`, `dhcpcd.conf` и `ntp.conf` соответственно.

Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

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

Сборка и запуск примера

См. "Сборка и запуск примеров".

Пример multi_vfs_dns_client

Этот пример показывает как использовать внешний dns-сервис в KasperskyOS.

Пример также демонстрирует использование разных виртуальных файловых систем (далее VFS) в одном решении:

- для работы с сетью используется программа `VfsNet`;
- для работы с файловой системой используется программа `VfsSdCardFs`.

Программа `Client` использует стандартные функции библиотеки `libc` для обращения ко внешнему dns-сервису, которые транслируются в обращения к программе `VfsNet` по IPC.

Для сборки и запуска примера используется система CMake из состава KasperskyOS Community Edition.

Поставляемые ресурсы

- В директории `./resources/edl` расположен файл `Client.edl`, который содержит статическое описание программы `Client`.
- В директории `./resources/hdd/etc` расположены файлы конфигурации для программ `VfsNet` и `Dhcpcd`: `hosts` и `dhcpcd.conf` соответственно.

Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

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

Сборка и запуск примера

См. "Сборка и запуск примеров".

Пример multi_vfs_dhcpcd

Пример использования программы `kl.rump.Dhcpcd`.

Программа `Dhcpcd` представляет собой реализацию DHCP-клиента, который в фоновом режиме получает параметры сетевых интерфейсов от внешнего DHCP-сервера и передает их виртуальной файловой системе (далее VFS).

Пример также демонстрирует использование разных VFS в одном решении. В примере для доступа к функциям работы с файловой системой и функциям работы с сетью используются разные VFS:

- Для работы с сетью используется программа `VfsNet`.
- Для работы с файловой системой используется программа `VfsSdCardFs`.

Программа `Client` использует стандартные функции библиотеки `libc` для получения информации о сетевых интерфейсах (`ioctl`), которые транслируются в обращения к VFS по IPC.

Для сборки и запуска примера используется система CMake из состава KasperskyOS Community Edition.

Поставляемые ресурсы

Директория `./resources/hdd/etc` содержит файлы конфигурации для программ VFS и `Dhcpcd`. Для конфигурации программы `Dhcpcd` используется стандартный синтаксис `dhcpcd.conf`.

В корневом файле `CMakeLists.txt` задаются значения переменных, которые определяют выбор файла конфигурации:

- `DHCPCD_FALLBACK`

  Динамическое получение параметров сетевых интерфейсов от внешнего DHCP-сервера с переходом на статическое задание параметров в случае недоступности DHCP-сервера. Значение используется по умолчанию.

- `DHCPCD_DYNAMIC`

  Динамическое получение параметров сетевых интерфейсов от внешнего DHCP-сервера.

- `DHCPCD_STATIC`

  Статическое задание параметров сетевых интерфейсов.

Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

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

Сборка и запуск примера

См. "Сборка и запуск примеров".

Пример mqtt_publisher (Mosquitto)

Пример использования протокола MQTT в KasperskyOS.

В этом примере MQTT-подписчик должен быть запущен в хостовой операционной системе, а MQTT-издатель в KasperskyOS. Программа `Publisher` представляет собой реализацию MQTT-издателя, который публикует текущее время с интервалом 5 секунд.

В результате успешного запуска и работы примера MQTT-подписчик, запущенный в хостовой операционной системе, выведет сообщение `"received PUBLISH"` с топиком `"datetime"`.

Пример также демонстрирует использование разных виртуальных файловых систем (далее VFS) в одном решении:

- для работы с сетью используется программа `VfsNet`;
- для работы с файловой системой используется программа `VfsSdCardFs`.

Для сборки и запуска примера используется система CMake из состава KasperskyOS Community Edition.

Запуск Mosquitto

Для запуска этого примера MQTT брокер Mosquitto должен быть установлен и запущен в хостовой системе. Для установки и запуска Mosquitto выполните следующие команды:

```
$ sudo apt install mosquitto mosquitto-clients
$ sudo /etc/init.d/mosquitto start
```

Для запуска MQTT-подписчика в хостовой системе выполните следующую команду:

```
$ mosquitto_sub -d -t "datetime"
```

Поставляемые ресурсы

- В директории `./resources/edl` расположен файл `Publisher.edl`, который содержит статическое описание программы `Publisher`.
- В директории `./resources/hdd/etc` расположены файлы конфигурации для программ `VfsNet`, `Dhcpcd` и `Ntpd`: `hosts`, `dhcpcd.conf` и `ntp.conf` соответственно.

Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

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

Сборка и запуск примера

См. "Сборка и запуск примеров".

Пример mqtt_subscriber (Mosquitto)

Пример использования протокола MQTT в KasperskyOS.

В этом примере MQTT-издатель должен быть запущен в хостовой операционной системе, а MQTT-подписчик в KasperskyOS. Программа `Subscriber` представляет собой реализацию MQTT-подписчика.

В результате успешного запуска и работы примера MQTT-подписчик, запущенный в KasperskyOS, выведет сообщение `"Got message with topic: my/awesome/topic, payload: hello"`.

Пример также демонстрирует использование разных виртуальных файловых систем (далее VFS) в одном решении:

- для работы с сетью используется программа `VfsNet`;
- для работы с файловой системой используется программа `VfsSdCardFs`.

Для сборки и запуска примера используется система CMake из состава KasperskyOS Community Edition.

Запуск Mosquitto

Для запуска этого примера MQTT брокер Mosquitto должен быть установлен и запущен в хостовой системе. Для установки и запуска Mosquitto выполните следующие команды:

```
$ sudo apt install mosquitto mosquitto-clients
$ sudo /etc/init.d/mosquitto start
```

Для запуска MQTT-издателя в хостовой системе выполните следующую команду:

```
$ mosquitto_pub  -t "my/awesome/topic" -m "hello"
```

Поставляемые ресурсы

- В директории `./resources/edl` расположен файл `Subscriber.edl`, который содержит статическое описание программы `Subscriber`.
- В директории `./resources/hdd/etc` расположены файлы конфигурации для программ `VfsNet`, `Dhcpcd` и `Ntpd`: `hosts`, `dhcpcd.conf` и `ntp.conf` соответственно.

Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

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

Сборка и запуск примера

См. "Сборка и запуск примеров".

Пример gpio_input

Пример использования драйвера GPIO.

Этот пример позволяет проверить функциональность ввода GPIO пинов. Используется порт "gpio0". Все пины, кроме указанных в массиве `exceptionPinArr`, по умолчанию ориентированы на ввод, напряжение на пинах согласуется с состоянием регистров подтягивающих резисторов. Состояния всех пинов, начиная с GPIO0 (с учетом указанных в массиве `exceptionPinArr`), будут последовательно считаны, сообщения о состояниях пинов будут выведены в консоль. Задержка между считываниями смежных пинов определяется макроопределением `DELAY_S` (время указывается в секундах).

`exceptionPinArr` - массив номеров GPIO пинов, которые необходимо исключить из примера. Это может понадобиться в случае, если часть пинов уже задействована для других функций, например, если пины используются для UART соединения при отладке.

При сборке и запуске этого примера на QEMU возникает ошибка. Это ожидаемое поведение, поскольку драйвера GPIO для QEMU нет.

При сборке и запуске этого примера на Raspberry Pi 4 B ошибка не возникает.

Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

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

Сборка и запуск примера

См. "Сборка и запуск примеров".

Пример gpio_output

Пример использования драйвера GPIO.

Этот пример позволяет проверить функциональность вывода GPIO пинов. Используется порт "gpio0". Начальное состояние всех пинов GPIO должно соответствовать логическому нулю (напряжение на пине отсутствует). Все пины, кроме указанных в массиве `exceptionPinArr`, будут настроены на вывод. Каждый пин, начиная с GPIO0 (с учетом указанных в массиве `exceptionPinArr`), будет последовательно переведен в состояние логической единицы (появление на пине напряжения), а затем в состояние логического нуля. Задержка между изменениями состояния пинов определяется макроопределением `DELAY_S` (время указывается в секундах). Включение/выключение пинов производится от `GPIO0` до `GPIO27` и обратно до `GPIO0`.

`exceptionPinArr` - массив номеров GPIO пинов, которые необходимо исключить из примера. Это может понадобиться в случае, если часть пинов уже задействована для других функций, например, если пины используются для UART соединения при отладке.

При сборке и запуске этого примера на QEMU возникает ошибка. Это ожидаемое поведение, поскольку драйвера GPIO для QEMU нет.

При сборке и запуске этого примера на Raspberry Pi 4 B ошибка не возникает.

Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

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

Сборка и запуск примера

См. "Сборка и запуск примеров".

Пример gpio_interrupt

Пример использования драйвера GPIO.

Этот пример позволяет проверить функциональность прерываний для GPIO пинов. Используется порт "gpio0". В битовой маске `pinsBitmap` структуры контекста прерываний `CallBackContext` пины из массива `exceptionPinArr` помечаются отработавшими, чтобы в дальнейшем пример мог корректно завершиться. Все пины, кроме указанных в массиве `exceptionPinArr`, переводятся в состояние `PINS_MODE`. Для всех пинов, кроме указанных в массиве `exceptionPinArr`, будет зарегистрирована функция обработки прерывания.

В бесконечном цикле происходит проверка условия равенства битовой маски `pinsBitmap` из структуры контекста прерываний `CallBackContext` битовой маске окончания работы примера `DONE_BITMASK` (соответствует условию, когда прерывание произошло на каждом GPIO пине). Также в цикле снимается функция-обработчик для последнего пина, на котором произошла обработка прерывания. При возникновении в первый раз прерывания на пине вызывается функция-обработчик, которая помечает соответствующий пин в битовой маске `pinsBitmap` в структуре контекста прерываний `CallBackContext` . Функция-обработчик для этого пина в дальнейшем снимается.

Следует учитывать возможное влияние начального состояния регистров подтягивающих резисторов для каждого пина на работу примера.

Прерывания для событий `GPIO_EVENT_LOW_LEVEL` и `GPIO_EVENT_HIGH_LEVEL` не поддерживаются.

`exceptionPinArr` - массив номеров GPIO пинов, которые необходимо исключить из примера. Это может понадобиться в случае, если часть пинов уже задействована для других функций, например, если пины используются для UART соединения при отладке.

При сборке и запуске этого примера на QEMU возникает ошибка. Это ожидаемое поведение, поскольку драйвера GPIO для QEMU нет.

При сборке и запуске этого примера на Raspberry Pi 4 B ошибка не возникает.

Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

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

Сборка и запуск примера

См. "Сборка и запуск примеров".

Пример gpio_echo

Пример использования драйвера GPIO.

Этот пример позволяет проверить функциональность ввода/вывода GPIO пинов, а также работу GPIO прерываний. Используется порт "gpio0". Пин вывода (`GPIO_PIN_OUT`) следует соединить с пином ввода (`GPIO_PIN_IN`). Устанавливается конфигурация для пина вывода (номер пина определяется в макросе `GPIO_PIN_OUT`), а также для пина ввода (`GPIO_PIN_IN`). Конфигурация пина ввода указана в макросе `IN_MODE`. Регистрируется обработчик прерываний для пина ввода. Несколько раз изменяется состояние пина вывода. В случае корректной работы примера, при изменении состояния пина вывода должен вызываться обработчик прерываний, который выводит состояние пина ввода, при этом состояния пина вывода и пина ввода должны совпадать.

При сборке и запуске этого примера на QEMU возникает ошибка. Это ожидаемое поведение, поскольку драйвера GPIO для QEMU нет.

При сборке и запуске этого примера на Raspberry Pi 4 B ошибка не возникает.

Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

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

Сборка и запуск примера

См. "Сборка и запуск примеров".

Пример koslogger

Пример демонстрирует использование библиотеки `spdlog` в KasperskyOS с помощью библиотеки-обертки `KOSLogger`.

В этом примере программа `Client` создает записи журнала, которые сохраняются на SD-карте (в случае запуска примера на Raspberry Pi) или в файле образа `build/einit/sdcard0.img` (при запуске примера на QEMU).

Пример также демонстрирует использование разных виртуальных файловых систем (далее VFS) в одном решении. В примере для доступа к функциям работы с файловой системой и функциям работы с сетью используются разные VFS:

- Для работы с сетью используется программа `VfsNet`.
- Для работы с файловой системой используется программа `VfsSdCardFs`.

Программа `kl.Ntpd` поставляется в составе KasperskyOS Community Edition и представляет собой реализацию ntp-клиента, который в фоновом режиме получает параметры времени от внешних ntp-серверов и передает их ядру KasperskyOS.

Программа `kl.rump.Dhcpcd` поставляется в составе KasperskyOS Community Edition и представляет собой реализацию DHCP-клиента, который в фоновом режиме получает параметры сетевых интерфейсов от внешнего DHCP-сервера и передает их виртуальной файловой системе.

Для сборки и запуска примера используется система `CMake` из состава KasperskyOS Community Edition.

Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

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

Сборка и запуск примера

См. "Сборка и запуск примеров".

Для корректной работы примера `koslogger` на Raspberry Pi после сборки примера и подготовки загрузочной SD-карты требуется выполнить следующие действия:

- создать директорию `/lib` на загрузочной SD-карте, если этой директории не существует;
- скопировать в директорию `/lib` на загрузочной SD-карте содержимое директории `build/hdd/lib`, которая генерируется во время сборки примера.

Пример pcre

Пример демонстрирует использование библиотеки `pcre` в KasperskyOS.

В этом примере программа `Client` использует библиотеку `pcre` и выводит результаты в консоль.

Для сборки и запуска примера используется система `CMake` из состава KasperskyOS Community Edition.

Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

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

Сборка и запуск примера

См. "Сборка и запуск примеров".

Для корректной работы примера `pcre` на Raspberry Pi после сборки примера и подготовки загрузочной SD-карты требуется выполнить следующие действия:

- создать директорию `/lib` на загрузочной SD-карте, если этой директории не существует;
- скопировать в директорию `/lib` на загрузочной SD-карте содержимое директории `build/hdd/lib`, которая генерируется во время сборки примера.

Пример messagebus

Пример демонстрирует использование компонента `MessageBus` в KasperskyOS.

В этом примере программы `Publisher` и `SubscriberA` и `SubscriberB` используют компонент MessageBus для обмена сообщениями.

Компонент `MessageBus` реализует шину сообщений. Программа `Publisher` является издателем и передает сообщения в шину. Программы `SubscriberA` и `SubscriberB` являются подписчиками и получают сообщения из шины.

Пример также демонстрирует использование разных виртуальных файловых систем (далее VFS) в одном решении. В примере для доступа к функциям работы с файловой системой и функциям работы с сетью используются разные VFS:

- Для работы с сетью используется программа `VfsNet`.
- Для работы с файловой системой используется программа `VfsSdCardFs`.

Программа `kl.Ntpd` поставляется в составе KasperskyOS Community Edition и представляет собой реализацию ntp-клиента, который в фоновом режиме получает параметры времени от внешних ntp-серверов и передает их ядру KasperskyOS.

Программа `kl.rump.Dhcpcd` поставляется в составе KasperskyOS Community Edition и представляет собой реализацию DHCP-клиента, который в фоновом режиме получает параметры сетевых интерфейсов от внешнего DHCP-сервера и передает их виртуальной файловой системе.

Для сборки и запуска примера используется система `CMake` из состава KasperskyOS Community Edition.

Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

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

Сборка и запуск примера

См. "Сборка и запуск примеров".

Для корректной работы примера `messagebus` на Raspberry Pi после сборки примера и подготовки загрузочной SD-карты требуется выполнить следующие действия:

- создать директорию `/lib` на загрузочной SD-карте, если этой директории не существует;
- скопировать в директорию `/lib` на загрузочной SD-карте содержимое директории `build/hdd/lib`, которая генерируется во время сборки примера.

Пример i2c_ds1307_rtc

Пример демонстрирует использование драйвера `i2c` (Inter-Integrated Circuit) в KasperskyOS.

В этом примере программа `I2cClient` использует интерфейс драйвера `i2c`.

Клиентская библиотека драйвера `i2c` статически компонуется с программой `I2cClient`. Реализация драйвера `i2c` использует подсистему BSP (Board Support Platform) для настройки частоты тактирования (Clocks) и мультиплексирование сигналов (PinMux). Поэтому, для корректной работы драйвера нужно:

- скомпоновать программу `I2cClient` с клиентской библиотекой `i2c_CLIENT_LIB`;
- скомпоновать программу `I2cClient` с клиентской библиотекой `bsp_CLIENT_LIB`;
- создать IPC-канал между программой `I2cClient` и драйвером `kl.drivers.I2C`;
- создать IPC-канал между программой `I2cClient` и драйвером `kl.drivers.BSP`.

Для сборки и запуска примера используется система `CMake` из состава KasperskyOS Community Edition.

Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

```
/opt/KasperskyOS-Community-Edition-<version>/examples/i2c_ds1307_rtc
```

Сборка и запуск примера

> Этот пример предназначен только для запуска на Raspberry Pi. Для корректной работы примера необходимо подключить к i2c порту модуль часов реального времени на микросхеме DS1307Z.

См. "Сборка и запуск примеров".

Пример iperf_separate_vfs

Пример демонстрирует использование библиотеки `iperf` в KasperskyOS.

В этом примере программа `Server` использует библиотеку `iperf`.

По умолчанию, в примере используется программная эмуляция (SLIRP) сети в QEMU. Если вы настроили TAP-интерфейсы для QEMU, то для корректной работы примера нужно изменить сетевые параметры запуска QEMU (переменная `QEMU_FLAGS`) в файле `einit/CMakeLists.txt` (подробнее см. комментарии в файле).

В примере не используется DHCP, поэтому IP-адрес сетевого интерфейса должен быть указан вручную в коде программы `Server` (`server/src/main.cpp`). SLIRP использует значения по умолчанию.

Библиотека `iperf` в примере используется в режиме сервера. Чтобы подключиться к этому серверу, установите программу `iperf3` на хостовой машине и запустите ее с помощью команды `iperf3 -c localhost`. Если вы настроили TAP-интерфейсы, укажите актуальный IP-адрес вместо `localhost`.

Первый запуск примера может занять продолжительное время, так как клиент `iperf` использует энтропию `/dev/urandom` для заполнения пакетов случайными данными. Чтобы избежать этого, запустите клиент `iperf` с параметром `--repeating-payload`.

Для сборки и запуска примера используется система `CMake` из состава KasperskyOS Community Edition.

Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

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

Сборка и запуск примера

См. "Сборка и запуск примеров".

Пример uart

Пример использования драйвера UART.

Этот пример показывает, как вывести сообщение "Hello world!" в соответствующий порт, используя драйвер UART.

При запуске эмуляции примера под QEMU, в флагах QEMU указывается `-serial stdio`. Это означает, что первый порт UART будет выводиться только в стандартный поток хостовой машины.

Полное описание интерфейса драйвера UART содержится в файле `/opt/KasperskyOS-Community-Edition-<version>/sysroot-aarch64-kos/include/uart/uart.h`.

Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

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

Сборка и запуск примера

См. "Сборка и запуск примеров".

Пример spi_check_regs

Пример демонстрирует использование драйвера `SPI` (Serial Peripheral Interface) в KasperskyOS.

Пример показывает как работать с интерфейсом SPI на плате расширения Sense HAT для Raspberry Pi. В этом примере программа `Client` использует интерфейс драйвера `SPI`. Программа открывает SPI-канал, выводит его параметры и выставляет нужный режим работы. После этого программа посылает по каналу последовательность данных и ожидает получения идентификатора контроллера ATTiny, установленного на плате Sense HAT.

Клиентская библиотека драйвера `SPI` статически компонуется с программой `Client`. Программа `Client` также использует драйвер `gpio` для установки режима работы контроллера и подсистему BSP (Board Support Platform) для настройки частоты тактирования (Clocks) и мультиплексирование сигналов (PinMux).

Для сборки и запуска примера используется система `CMake` из состава KasperskyOS Community Edition.

Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

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

Сборка и запуск примера

> Этот пример предназначен только для запуска на Raspberry Pi. Для корректной работы примера необходимо подключить к SPI порту модуль Sense HAT.

См. "Сборка и запуск примеров".

Пример barcode_scanner

Пример демонстрирует использование драйвера `USB` (Universal Serial Bus) в KasperskyOS с помощью библиотеки `libevdev`.

В этом примере программа `BarcodeScanner` использует библиотеку `libevdev` для взаимодействия со сканером штрихкодов, подключенным к USB порту Raspberry Pi.

Программа ожидает сигналов от сканера штрихкодов и выводит полученные данные в `stderr`.

Для сборки и запуска примера используется система `CMake` из состава KasperskyOS Community Edition.

Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

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

Сборка и запуск примера

> Этот пример предназначен только для запуска на Raspberry Pi. Для корректной работы примера необходимо подключить к USB порту сканер штрихкодов, работающий в режиме эмуляции клавиатуры (например Zebra Symbol LS2208).

См. "Сборка и запуск примеров".

Пример perfcnt

Пример демонстрирует использование счетчиков производительности в KasperskyOS.

Пример включает в себя две программы: `Worker` и `Monitor`.

Программа `Worker` выполняет вычисления в цикле, периодически нагружая процессор и используя память.

Программа `Monitor` использует функцию `KnProfilerGetCounter()` библиотеки `libkos` для получения значений счетчиков производительности для программы `Worker` и выводит их в консоль.

Для сборки и запуска примера используется система `CMake` из состава KasperskyOS Community Edition.

> При сборке и запуске этого примера на QEMU некоторые счетчики производительности могут работать некорректно.

Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

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

Сборка и запуск примера

См. "Сборка и запуск примеров".

Пример watchdog_system_reset

Пример демонстрирует использование драйвера `Watchdog` в KasperskyOS.

В этом примере программа `Client` использует интерфейс драйвера `Watchdog` для взаимодействия со сторожевым таймером:

- получает текущие параметры драйвера `Watchdog` и выводит их в `stderr`;
- изменяет значение таймера по умолчанию на новое и запускает таймер;
- несколько раз сбрасывает таймер;
- ожидает перезагрузки системы при срабатывании таймера.

Клиентская библиотека драйвера `Watchdog` статически компонуется с программой `Client`.

Для сборки и запуска примера используется система `CMake` из состава KasperskyOS Community Edition.

Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

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

Сборка и запуск примера

См. "Сборка и запуск примеров".

Пример shared_libs

Пример демонстрирует использование статических и динамических библиотек в KasperskyOS.

В примере программа `Client` выполняет следующие действия:

- вызывает функцию из статической библиотеки `hello_s`;
- вызывает функцию из динамической библиотеки `hello_d1`, скомпонованной вместе с программой и загружаемой в память при запуске процесса;
- вызывает функцию из динамической библиотеки `hello_d2`, загружаемой в память при вызове функции `dlopen()` интерфейса POSIX.

Чтобы динамические библиотеки могли быть разделяемыми между разными процессами, в пример включена системная программа `BlobContainer`.

Для сборки и запуска примера используется система `CMake` из состава KasperskyOS Community Edition.

Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

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

Сборка и запуск примера

См. "Сборка и запуск примеров".

Для корректной работы примера `shared_libs` на Raspberry Pi после сборки примера и подготовки загрузочной SD-карты требуется выполнить следующие действия:

- создать директорию `/lib` на загрузочной SD-карте, если этой директории не существует;
- скопировать в директорию `/lib` на загрузочной SD-карте содержимое директории `build/hdd/lib`, которая генерируется во время сборки примера.

Сведения о некоторых лимитах, установленных в системе

Заголовочные и IDL-файлы из состава KasperskyOS SDK содержат константы, устанавливающие лимиты в системе (см. таблицу ниже).

Константы, устанавливающие лимиты в системе

Подсистема	Константы
`POSIX`	Константы в файле `sysroot-*-kos/include/limits.h`.
`BlobContainer`	Константы в файлах `sysroot-*-kos/include/kl/EntityLauncher.idl(.h)`: • `MaxArgSize (kl_EntityLauncher_MaxArgSize)` – максимальный размер параметра запуска программы и переменной окружения, в байтах; • `MaxArgsCount (kl_EntityLauncher_MaxArgsCount)` – максимальное число параметров запуска программы и максимальное число переменных окружения для программы.
`CertificateStorage`	Константы в файлах `sysroot-*-kos/include/kl/CertificateStorage.idl(.h)`: • `MaxNumCerts (kl_CertificateStorage_MaxNumCerts)` – максимальное число сертификатов в хранилище; • `MaxCertSize (kl_CertificateStorage_MaxCertSize)` – максимальный размер сертификата, в байтах; • `HashSize (kl_CertificateStorage_HashSize)` – размер хеша хранилища сертификатов, в байтах.

Tls	Константы в файлах sysroot-*-kos/include/kl/CertificatePolicy.idl(.h): • MaxDERCertDataSize (kl_CertificatePolicy_MaxDERCertDataSize) – максимальный размер сертификата в формате DER, в байтах; • MaxHostAddressBufferSize (kl_CertificatePolicy_MaxHostAddressBufferSize) – максимальный размер буфера для адреса хоста, в байтах. Константы в файлах sysroot-*-kos/include/kl/crypto/tls/TlsEvent.idl(.h): • FunctionNameSize (kl_crypto_tls_TlsEvent_FunctionNameSize) – максимальная длина имени функции, в байтах; • IdSize (kl_crypto_tls_TlsEvent_IdSize) – размер идентификатора сессии, в байтах; • HostnameSize (kl_crypto_tls_TlsEvent_HostnameSize) – максимальная длина имени хоста, в байтах; • PkiEntrySize (kl_crypto_tls_TlsEvent_PkiEntrySize) – максимальный размер PKI-сертификата, в байтах; • MaxCertificatesInChain (kl_crypto_tls_TlsEvent_MaxCertificatesInChain) – максимальное число сертификатов в цепочке; • MaxCertificatesInTrustedSet (kl_crypto_tls_TlsEvent_MaxCertificatesInTrustedSet) – максимальное число доверенных сертификатов; • KeyFingerprintLength (kl_crypto_tls_TlsEvent_KeyFingerprintLength) – размер отпечатка ключа, в байтах; • MbedTlsDescriptionSize (kl_crypto_tls_TlsEvent_MbedTlsDescriptionSize) – максимальный размер описания ошибки в MbedTLS, в байтах; • VfsDescriptionSize (kl_crypto_tls_TlsEvent_VfsDescriptionSize) – максимальный размер описания ошибки в VFS, в байтах; • DescriptionSize (kl_crypto_tls_TlsEvent_DescriptionSize) – максимальный размер описания события, в байтах.
ExecutionManager	Константы в файлах sysroot-*-kos/include/kl/execution_manager/Types.idl(.h): • NkAppNameMaxSize (kl_execution_manager_Types_NkAppNameMaxSize) – максимальная длина имени программы, в байтах;

	• NkPathMaxSize (kl_execution_manager_Types_NkPathMaxSize) – максимальная длина пути к исполняемому файлу, в байтах; • NkEntityNameMaxSize (kl_execution_manager_Types_NkEntityNameMaxSize) – максимальная длина имени процесса, в байтах; • NkEiidMaxSize (kl_execution_manager_Types_NkEiidMaxSize) – максимальная длина имени класса процессов, в байтах; • NkTaskNameMaxSize (kl_execution_manager_Types_NkTaskNameMaxSize) – максимальная длина имени процесса, в байтах; • NkArgMaxLen (kl_execution_manager_Types_NkArgMaxLen) – максимальный размер параметра запуска программы, в байтах; • NkEnvMaxLen (kl_execution_manager_Types_NkEnvMaxLen) – максимальный размер переменной окружения, в байтах; • NkArgsArrayMaxSize (kl_execution_manager_Types_NkArgsArrayMaxSize) – максимальное число параметров запуска программы; • NkEnvsArrayMaxSize (kl_execution_manager_Types_NkEnvsArrayMaxSize) – максимальное число переменных окружения для программы.
KlogStorage	Константы в файлах sysroot-*-kos/include/kl/KlogStorage.idl(.h): • StringSize (kl_KlogStorage_StringSize) – максимальный размер сообщения, в байтах; • MaxMessages (kl_KlogStorage_MaxMessages) – максимальное число сообщений.
Env	Константы в файлах sysroot-*-kos/include/kl/Env.idl(.h): • MaxArgsCount (kl_Env_MaxArgsCount) – максимальное число параметров запуска программы и максимальное число переменных окружения для программы; • MaxArgSize (kl_Env_MaxArgSize) – максимальный размер параметра запуска программы и переменной окружения, в байтах; • MaxNameSize (kl_Env_MaxNameSize) – максимальная длина имени процесса, в байтах.
VFS	Константы в файлах sysroot-*-kos/include/kl/VfsTypes.idl(.h): • MaxBytesCount (kl_VfsTypes_MaxBytesCount) – максимальный размер буфера для передачи данных в VFS, в байтах; • MaxPathSize (kl_VfsTypes_MaxPathSize) – максимальная длина пути, в байтах;

- `MaxDevnameSize (kl_VfsTypes_MaxDevnameSize)` – максимальная длина имени устройства, в байтах;
- `MaxFstypeSize (kl_VfsTypes_MaxFstypeSize)` – максимальная длина имени файловой системы, в байтах;
- `MaxFsDataSize (kl_VfsTypes_MaxFsDataSize)` – максимальный размер данных для параметра `data` функции `mount()`, в байтах;
- `MaxFcntlTSize (kl_VfsTypes_MaxFcntlTSize)` – максимальный размер данных для опционального параметра функции `fcntl()`, в байтах;
- `MaxIoctlTSize (kl_VfsTypes_MaxIoctlTSize)` – максимальный размер данных для опционального параметра функции `ioctl()`, в байтах;
- `MaxSockAddrSize (kl_VfsTypes_MaxSockAddrSize)` – максимальный размер IP-адреса, в байтах;
- `MaxSockOptionSize (kl_VfsTypes_MaxSockOptionSize)` – максимальный размер данных для параметра `option_value` функций `getsockopt()` и `setsockopt()`, в байтах;
- `MaxHostnameSize (kl_VfsTypes_MaxHostnameSize)` – максимальная длина имени хоста, в байтах;
- `MaxServnameSize (kl_VfsTypes_MaxServnameSize)` – максимальный размер данных для параметра `servname` функции `getaddrinfo()` и параметра `service` функции `getnameinfo()`, в байтах;
- `MaxMsgNameSize (kl_VfsTypes_MaxMsgNameSize)` – максимальный размер данных для элемента `msg_name` параметра `message` функций `recvmsg()` и `sendmsg()`, в байтах;
- `MaxMsgDataSize (kl_VfsTypes_MaxMsgDataSize)` – максимальный размер данных для элемента `msg_control` параметра `message` функций `recvmsg()` и `sendmsg()`, в байтах;
- `MaxIovDataSize (kl_VfsTypes_MaxIovDataSize)` – максимальный размер буфера, описываемого структурой `iovec` в параметре `message` функций `recvmsg()` и `sendmsg()` и в параметре `iov` функций `readv()` и `writev()`, в байтах;
- `MaxIovecsCount (kl_VfsTypes_MaxIovecsCount)` – максимальное число структур `iovec` в параметре `message` функций `recvmsg()` и `sendmsg()` и в параметре `iov` функций `readv()` и `writev()`;
- `MaxAddrinfoSize (kl_VfsTypes_MaxAddrinfoSize)` – максимальный размер данных для параметра `res` функции `getaddrinfo()`, в байтах;
- `VfsHostent (kl_VfsTypes_MaxHostentSize)` – максимальный размер данных для возвращаемого значения функции `gethostbyname()`, в байтах;
- `VfsDnsName (kl_VfsTypes_MaxDnsNameSize)` – максимальный размер данных для параметра `name` функции `getnetbyname()`, в байтах;

	• `MaxProtoentNameSize (kl_VfsTypes_MaxProtoentNameSize)` – максимальная длина имени протокола в параметре `name` функции `getprotobyname()`, а также в возвращаемом значении функций `getprotobyname()` и `getprotobynumber()`, в байтах; • `MaxProtoentAliasesSize (kl_VfsTypes_MaxProtoentAliasesSize)` – максимальная длина псевдонима протокола в возвращаемом значении функций `getprotobyname()` и `getprotobynumber()`, в байтах.
`MessageBus`	Константы в файлах `sysroot-*-kos/include/kl/MessageBusTypes.idl(.h)`: • `MaxStringLength (kl_MessageBusTypes_MaxStringLength)` – максимальный размер сообщения, в байтах.
`Dhcpcd`	Константы в файлах `sysroot-*-kos/include/kl/rump/DhcpcdConfig.idl(.h)`: • `MaxDhcpcdStrSize (kl_rump_DhcpcdConfig_MaxDhcpcdStrSize)` – максимальный размер набора параметров, получаемых от DHCP-сервера, в байтах.
`Terminal`	Константы в файлах `sysroot-*-kos/include/kl/Terminal.idl(.h)`: • `MaxTerminalBytesCount (kl_Terminal_MaxTerminalBytesCount)` – максимальный размер буфера для записи в терминал и чтения из терминала, в байтах; • `MaxTerminalConnectionIdSize (kl_Terminal_MaxTerminalConnectionIdSize)` – максимальный размер идентификатора терминала, в байтах.

Лицензирование

Лицензионное соглашение – это юридическое соглашение между вами и АО "Лаборатория Касперского", в котором указано, на каких условиях вы можете использовать KasperskyOS Community Edition.

> Внимательно ознакомьтесь с условиями Лицензионного соглашения перед началом работы с KasperskyOS Community Edition.

Вы можете ознакомиться с условиями Лицензионного соглашения следующими способами:

- Прочитав текст Лицензионного соглашения перед скачиванием дистрибутива KasperskyOS Community Edition.
- Прочитав документ EULA.<код языка>.txt, расположенный в директории `/opt/KasperskyOS-Community-Edition-<version>` после установки KasperskyOS Community Edition.

Вы принимаете условия Лицензионного соглашения, устанавливая флажок **Согласен** под текстом Лицензионного соглашения перед скачиванием дистрибутива KasperskyOS Community Edition.

Если вы не согласны с условиями Лицензионного соглашения, вы должны прервать скачивание дистрибутива и не должны использовать KasperskyOS Community Edition.

Предоставление данных

Версии KasperskyOS Community Edition

KasperskyOS Community Edition поставляется в двух версиях:

- версия, которую можно загрузить с русскоязычного сайта https://os.kaspersky.ru/development.
- версия, которую можно загрузить с англоязычного сайта https://os.kaspersky.com/development.

Версии различаются содержимым файла Лицензионного соглашения, а также информацией, передаваемой на серверы "Лаборатории Касперского" в автоматическом режиме при сборке решения с использованием CMake-библиотек из состава SDK.

Предоставление данных в KasperskyOS Community Edition

Версия KasperskyOS Community Edition, загруженная с русскоязычного сайта, при запуске сборки решения в автоматическом режиме передает на серверы "Лаборатории Касперского" следующую информацию:

- номер установленной версии KasperskyOS Community Edition;
- уникальный идентификатор оборудования, представляющий собой хеш-сумму даты создания директории /opt/KasperskyOS-Community-Edition-<version>.

Версия KasperskyOS Community Edition, загруженная с англоязычного сайта, при запуске сборки решения в автоматическом режиме передает на серверы "Лаборатории Касперского" следующую информацию:

- номер установленной версии KasperskyOS Community Edition.

Для обеих версий, помимо передачи данных, в процессе сборки решения также проходит проверка наличия более новой версии KasperskyOS Community Edition.

> При отсутствии соединения с интернетом, сборка решения происходит без передачи данных и проверки наличия обновлений.

Вы можете отключить проверку наличия обновлений SDK и передачу данных версии SDK на сервер Kaspersky, используя параметр `NO_NEW_VERSION_CHECK` CMake-команды `initialize_platform()` при сборке решения.

Данные передаются с целью учета количества пользователей KasperskyOS Community Edition и получения информации о распространении и использовании KasperskyOS Community Edition.

Полученная информация защищается "Лабораторией Касперского" в соответствии с установленными законом требованиями и действующими правилами "Лаборатории Касперского". Данные передаются по зашифрованным каналам связи.

Глоссарий

Callable-дескриптор

Callable-дескриптор (англ. callable handle) – это клиентский IPC-дескриптор, который идентифицирует одновременно IPC-канал до сервера и службу этого сервера.

Связанные разделы:
Создание дескрипторов

CDL

Component Definition Language – декларативный язык для создания формальной спецификации компонента решения.

Связанные разделы:
Формальные спецификации компонентов решения на базе KasperskyOS
CDL-описание

DMA

Прямой доступ к памяти

EDL

Entity Definition Language – декларативный язык для создания формальной спецификации компонента решения.

Связанные разделы:
Формальные спецификации компонентов решения на базе KasperskyOS
EDL-описание

Execute-интерфейс

Интерфейс, через который ядро KasperskyOS обращается к модулю безопасности Kaspersky Security Module, чтобы сообщить о запуске ядра или об инициации запуска процесса ядром или другим процессом.

Связанные разделы:
Установка глобальных параметров политики безопасности решения на базе KasperskyOS
Привязка методов моделей безопасности к событиям безопасности

IDL

Interface Definition Language – декларативный язык для создания формальной спецификации компонента решения.

Связанные разделы:
Формальные спецификации компонентов решения на базе KasperskyOS
IDL-описание

Init-описание

Init-описание представляет собой текстовый файл, содержащий данные в формате YAML, идентифицирующие процессы и IPC-каналы, которые создаются при запуске решения. Файл init-описания обычно имеет имя `init.yaml`.

Связанные разделы:
Обзор: Einit и init.yaml
Примеры init-описаний
Шаблон init.yaml.in

IPC

Межпроцессное взаимодействие

IPC-дескриптор

IPC-дескриптор (англ. IPC handle) – это дескриптор, который идентифицирует IPC-канал. Клиентский IPC-дескриптор нужен для выполнения системного вызова `Call()`. Серверный IPC-дескриптор требуется для выполнения системных вызовов `Recv()` и `Reply()`. Callable-дескриптор и слушающий дескриптор являются IPC-дескрипторами.

Связанные разделы:
Механизм IPC
Создание дескрипторов
Создание IPC-каналов

IPC-запрос

IPC-сообщение, отправляемое серверу клиентом.

Связанные разделы:
Механизм IPC

IPC-канал

Объект ядра KasperskyOS, который позволяет процессам взаимодействовать друг с другом, передавая IPC-сообщения. IPC-канал имеет клиентскую и серверную стороны, которые идентифицируются клиентским и серверным IPC-дескриптором соответственно.

Связанные разделы:
Механизм IPC
Создание IPC-каналов

IPC-ответ

IPC-сообщение, отправляемое клиенту сервером.

Связанные разделы:
Механизм IPC

IPC-сообщение

Пакет данных, который передается между процессами, а также между процессами и ядром KasperskyOS для осуществления IPC. IPC-сообщение содержит фиксированную часть и опционально арену.

Связанные разделы:
Обзор: структура IPC-сообщения

IPC-транспорт

Надстройка над системными вызовами отправки и приема IPC-сообщений, которая позволяет отдельно работать с фиксированной частью и ареной IPC-сообщений. Поверх этой надстройки работает транспортный код.

Связанные разделы:
Инициализация IPC-транспорта для межпроцессного взаимодействия и управление обработкой IPC-запросов (transport-kos.h, transport-kos-dispatch.h)
Инициализация IPC-транспорта для обращения к модулю безопасности (transport-kos-security.h)

KasperskyOS

Специализированная операционная система на основе микроядра разделения и монитора безопасности.

Связанные разделы:
Общие сведения

KSM

Kaspersky Security Module – модуль ядра KasperskyOS, который разрешает или запрещает взаимодействия процессов между собой и с ядром, а также обрабатывает обращения процессов через интерфейс безопасности.

Связанные разделы:
Управление IPC
Управление доступом к ресурсам

KSS

Kaspersky Security System – технология, позволяющая реализовать политики безопасности решений. Эта технология предусматривает создание формальных спецификаций компонентов решений и описаний политик безопасности решений с использование моделей безопасности.

Связанные разделы:
Общие сведения
Разработка политик безопасности

MID

Идентификатор метода службы

OCap

Object Capability – механизм безопасности на основе мандатных ссылок.

Связанные разделы:
Управление доступом к ресурсам
Управление дескрипторами (handle_api.h)

OPP

Точка рабочих характеристик

PAL

Policy Assertion Language – декларативный язык для создания тестов политики безопасности решения.

Связанные разделы:
Создание и выполнение тестов политики безопасности решения на базе KasperskyOS
Примеры тестов политик безопасности решений на базе KasperskyOS

PSL

Policy Specification Language – декларативный язык для создания описания политики безопасности решения.

Связанные разделы:
Описание политики безопасности решения на базе KasperskyOS
Синтаксис языка PSL

RIID

Идентификатор службы

SID

Идентификатор безопасности

Аппаратное прерывание

Сигнал процессору от устройства о необходимости немедленно переключиться с исполнения текущей программы на обработку события, связанного с этим устройством. Например, нажатие клавиши на клавиатуре вызывает аппаратное прерывание, которое обеспечивает требуемую реакцию на это нажатие (к примеру, ввод символа).

Связанные разделы:
Управление обработкой прерываний (irq.h)

Арена IPC-сообщения

Опциональная часть IPC-сообщения, которая содержит параметры интерфейсных методов (и/или элементы этих параметров) переменного размера.

Связанные разделы:
Обзор: структура IPC-сообщения
Работа с ареной IPC-сообщений

Аудит безопасности

Аудит безопасности представляет собой следующую последовательность действий. Модуль безопасности Kaspersky Security Module сообщает ядру KasperskyOS сведения о решениях, принятых этим модулем. Затем ядро передает эти данные системной программе `Klog`, которая декодирует их и передает системной программе `KlogStorage` (передача данных осуществляется через IPC). Последняя направляет полученные данные в стандартный вывод (или стандартный вывод ошибок) либо записывает в файл.

Связанные разделы:
Создание профилей аудита безопасности
Примеры профилей аудита безопасности
Использование системных программ Klog и KlogStorage для выполнения аудита безопасности

Барьер памяти

Барьер памяти (англ. memory barrier) – это инструкция для компилятора и процессора, которая гарантирует, что операции доступа к памяти, указанные в исходном коде до установки барьера, будут выполнены до операций доступа к памяти, указанных в исходном коде после установки барьера.

Связанные разделы:
Использование барьеров памяти (barriers.h)

Блокировка чтения-записи

Примитив синхронизации, который используется, чтобы разрешить доступ к разделяемым между потоками исполнения ресурсам либо на запись для одного потока исполнения, либо на чтение для нескольких потоков исполнения одновременно.

Связанные разделы:
Использование примитивов синхронизации (event.h, mutex.h, rwlock.h, semaphore.h, condvar.h)

Буфер DMA

Буфер, состоящий из одного или нескольких регионов физической памяти (блоков), используемых для прямого доступа к памяти.

Связанные разделы:
Использование DMA (dma.h)
Управление изоляцией памяти для ввода-вывода (iommu_api.h)

Выражение модели безопасности

Метод модели безопасности, возвращающий значения, которые могут использоваться как входные данные для других методов моделей безопасности.

Связанные разделы:
Общие сведения об описании политики безопасности решения на базе KasperskyOS
Привязка методов моделей безопасности к событиям безопасности
Модели безопасности KasperskyOS

Данные аудита безопасности

Сведения о решениях модуля безопасности Kaspersky Security Module, которые включают сами решения ("разрешено" или "запрещено"), описания событий безопасности, результаты вызовов методов моделей безопасности, а также данные о некорректности IPC-сообщений.

Связанные разделы:
Создание профилей аудита безопасности

Дерево наследования дескрипторов

Иерархия порождения дескрипторов ресурса, которая хранится в ядре KasperskyOS.

Связанные разделы:
Управление дескрипторами (handle_api.h)

Дескриптор

Дескриптор (англ. handle) – это идентификатор ресурса (например, участка памяти, порта, сетевого интерфейса, IPC-канала). Дескриптор IPC-канала называется IPC-дескриптором.

Связанные разделы:
Управление доступом к ресурсам
Управление дескрипторами (handle_api.h)

Также в KasperskyOS используются другие дескрипторы:

- дескриптор арены;
- дескриптор участка арены.

Дескриптор арены

Структура, содержащая три указателя: на начало арены, на начало неиспользованной части арены и на конец арены.

Связанные разделы:
Работа с ареной IPC-сообщений

Дескриптор участка арены

Структура, содержащая смещение участка арены в байтах (относительно начала арены) и размер участка арены в байтах.

Связанные разделы:
Работа с ареной IPC-сообщений

Идентификатор безопасности

Идентификатор безопасности (англ. Security Identifier, SID) – это глобальный уникальный идентификатор ресурса. Модуль безопасности Kaspersky Security Module идентифицирует ресурсы по их идентификаторам безопасности.

Связанные разделы:
Управление доступом к ресурсам
Получение идентификатора безопасности (SID)

Идентификатор метода службы

Идентификатор метода службы (англ. Method Identifier, MID) – это порядковый номер метода службы в наборе методов этой службы (начиная с нуля).

Связанные разделы:
Механизм IPC
Обзор: структура IPC-сообщения

Идентификатор службы

Идентификатор службы (англ. Runtime Implementation Identifier, RIID) – это порядковый номер службы в наборе служб сервера (начиная с нуля).

Связанные разделы:
Механизм IPC
Обзор: структура IPC-сообщения

Инициализирующая программа

Программа `Einit`, которая запускается ядром KasperskyOS и в соответствии с init-описанием запускает другие программы и создает IPC-каналы.

Связанные разделы:
Обзор: Einit и init.yaml
Файл CMakeLists.txt для сборки программы Einit
Структура и запуск решения на базе KasperskyOS
einit

Интерфейс безопасности

Интерфейс, который используется для взаимодействия процесса с модулем безопасности Kaspersky Security Module. Интерфейс безопасности определяется в IDL-описании.

Связанные разделы:
Формальные спецификации компонентов решения на базе KasperskyOS
EDL-описание
CDL-описание
IDL-описание
Привязка методов моделей безопасности к событиям безопасности
Инициализация IPC-транспорта для обращения к модулю безопасности (transport-kos-security.h)

Интерфейс службы

Набор сигнатур методов службы. Интерфейс службы определяется в IDL-описании.

Связанные разделы:
Формальные спецификации компонентов решения на базе KasperskyOS
IDL-описание

Интерфейсный метод

Подпрограмма, вызываемая через IPC.

Связанные разделы:
Механизм IPC
IDL-описание
Методы служб ядра KasperskyOS

Клиент

В контексте IPC *клиент* – клиентский процесс.

Клиентская библиотека компонента решения

Транспортная библиотека, которая преобразует локальные вызовы в IPC-запросы.

Связанные разделы:
Транспортный код для IPC

Клиентский процесс

Процесс, использующий службу другого процесса с помощью механизма IPC. Один процесс может использовать одновременно несколько IPC-каналов. При этом для одних IPC-каналов процесс может быть клиентом, а для других IPC-каналов этот же процесс может быть сервером.

Связанные разделы:
Механизм IPC

Компонент решения на базе KasperskyOS

Программа, входящая в решение.

Связанные разделы:
Общие сведения
Формальные спецификации компонентов решения на базе KasperskyOS

Контекст безопасности

Данные, ассоциированные с идентификатором безопасности, которые используются модулем безопасности Kaspersky Security Module для принятия решений.

Связанные разделы:
Управление доступом к ресурсам

Контекст передачи ресурса

Данные, позволяющие серверу идентифицировать ресурс и его состояние, когда запрашивается доступ к ресурсу через потомков переданного дескриптора. В общем случае это набор разнотипных данных (структура). Например, для файла контекст передачи может включать имя, путь, положение курсора.

Связанные разделы:
Управление дескрипторами (handle_api.h)

Контекст пользовательского ресурса

Данные, позволяющие поставщику ресурса идентифицировать ресурс и его состояние, когда запрашивается доступ к ресурсу другими процессами. В общем случае это набор разнотипных данных (структура). Например, для файла контекст может включать имя, путь, положение курсора.

Связанные разделы:
Управление дескрипторами (handle_api.h)

Конфигурация аудита безопасности

Элемент профиля аудита безопасности, который задает объекты моделей безопасности, покрываемые аудитом безопасности, а также условия выполнения аудита безопасности.

Связанные разделы:
Создание профилей аудита безопасности
Примеры профилей аудита безопасности

Критическая секция

Участок кода, в котором осуществляется обращение к разделяемым между потоками исполнения ресурсам.

Связанные разделы:
Использование примитивов синхронизации (event.h, mutex.h, rwlock.h, semaphore.h, condvar.h)

Мандатная ссылка

Каждый дескриптор ассоциируется с правами доступа к идентифицируемому им ресурсу, то есть является мандатной ссылкой (англ. capability) в терминах механизма безопасности на основе мандатных ссылок (англ. Object Capability, OCap). Получая дескриптор, процесс получает права доступа к ресурсу, который этот дескриптор идентифицирует. Например, правами доступа могут быть: право на чтение, право на запись, право на передачу другому процессу возможности выполнять операции над ресурсом (право на передачу дескриптора).

Связанные разделы:
Управление доступом к ресурсам
Управление дескрипторами (handle_api.h)

Маска прав дескриптора

Значение, биты которого интерпретируются как права доступа к ресурсу, который идентифицируется дескриптором.

Связанные разделы:
Управление доступом к ресурсам
Маска прав дескриптора
Управление дескрипторами (handle_api.h)

Маска событий

Значение, биты которого интерпретируются как события, которые должны отслеживаться или уже произошли. Маска событий имеет размер 32 бита и состоит из общей и специальной части. Общая часть описывает события, неспецифичные для любых ресурсов. Специальная часть описывает события, специфичные для ресурсов.

Связанные разделы:
Использование уведомлений (notice_api.h)
Передача дескрипторов

Межпроцессное взаимодействие

Межпроцессное взаимодействие (англ. Inter-Process Communication, IPC) – механизм взаимодействия процессов друг с другом и с ядром KasperskyOS.

Связанные разделы:
IPC
Инициализация IPC-транспорта для межпроцессного взаимодействия и управление обработкой IPC-запросов (transport-kos.h, transport-kos-dispatch.h)
Ограничения поддержки POSIX

Метод модели безопасности

Элемент модели безопасности, который определяет допустимость взаимодействий процессов между собой и с ядром KasperskyOS.

Связанные разделы:
Общие сведения об описании политики безопасности решения на базе KasperskyOS
Привязка методов моделей безопасности к событиям безопасности
Примеры привязок методов моделей безопасности к событиям безопасности
Модели безопасности KasperskyOS

Метод службы

Интерфейсный метод

Модель безопасности KasperskyOS

Фреймворк для реализации политик безопасности решений.

Связанные разделы:
Описание политики безопасности решения на базе KasperskyOS
Модели безопасности KasperskyOS

Мьютекс

Примитив синхронизации, который обеспечивает взаимоисключающее исполнение критических секций.

Связанные разделы:
Использование примитивов синхронизации (event.h, mutex.h, rwlock.h, semaphore.h, condvar.h)
Ограничения поддержки POSIX

Начальное значение генератора случайных чисел

Начальное значение генератора случайных чисел (англ. seed) – значение, которое определяет последовательность генерируемых случайных чисел. То есть после установки одного и того же начального значения генератор создает одинаковые последовательности случайных чисел. (Энтропия таких чисел полностью определяется энтропией начального значения, поэтому точнее их называть не случайными, а псевдослучайными.)

Связанные разделы:
Генерация случайных чисел (random_api.h)

Объект контекста передачи ресурса

Объект ядра KasperskyOS, в котором хранится указатель на контекст передачи ресурса.

Связанные разделы:
Управление дескрипторами (handle_api.h)

Объект модели безопасности

Экземпляр класса, определение которого является формальным описанием модели безопасности (в PSL-файле).

Связанные разделы:
Общие сведения об описании политики безопасности решения на базе KasperskyOS
Создание объектов моделей безопасности
Модели безопасности KasperskyOS

Описание политики безопасности решения на базе KasperskyOS

Набор связанных между собой текстовых файлов с расширением `psl`, которые содержат декларации на языке PSL.

Связанные разделы:
Описание политики безопасности решения на базе KasperskyOS
Общие сведения об описании политики безопасности решения на базе KasperskyOS
Шаблон security.psl.in
Примеры описаний простейших политик безопасности решений на базе KasperskyOS
Методы служб ядра KasperskyOS

Паттерн безопасности

Паттерн безопасности (также шаблон безопасности) описывает конкретную повторяющуюся проблему безопасности, которая возникает в определенных известных контекстах, а также предлагает хорошо зарекомендовавшую себя общую схему решения такой проблемы безопасности. Паттерн это не законченный проект, который можно преобразовать непосредственно в код, а решение общей проблемы, встречающейся в различных проектах.

Связанные разделы:
Паттерны безопасности при разработке под KasperskyOS

Политика безопасности решения на базе KasperskyOS

Логика обработки событий безопасности в решении, реализуемая модулем безопасности Kaspersky Security Module. Исходный код модуля безопасности Kaspersky Security Module генерируется из описания политики безопасности решения и формальных спецификаций компонентов решения.

Связанные разделы:
Общие сведения

Пользовательский ресурс

Ресурс, которым управляет процесс. Примеры пользовательских ресурсов: файлы, устройства ввода-вывода, накопители данных.

Связанные разделы:
Управление доступом к ресурсам
Управление дескрипторами (handle_api.h)

Поставщик ресурсов

Процесс, который управляет пользовательскими ресурсами и доступом к этим ресурсам для других процессов. Поставщиками ресурсов являются, например, драйверы.

Связанные разделы:
Управление доступом к ресурсам
Управление дескрипторами (handle_api.h)

Поток исполнения

Поток исполнения (англ. thread) – это абстракция, используемая для управления исполнением кода программы. Один процесс может включать один или несколько потоков исполнения. Каждому потоку исполнения отдельно выделяется процессорное время. В одном потоке может исполняться весь код программы или только часть. Один и тот же код программы может исполняться в нескольких потоках.

Связанные разделы:
Ограничения поддержки POSIX

Потребитель ресурсов

Процесс, который использует ресурсы, предоставляемые ядром KasperskyOS или другими процессами.

Связанные разделы:
Управление доступом к ресурсам
Управление дескрипторами (handle_api.h)
Модель безопасности Mic

Правило модели безопасности

Метод модели безопасности, возвращающий результат "разрешено" или "запрещено".

Связанные разделы:
Общие сведения об описании политики безопасности решения на базе KasperskyOS
Привязка методов моделей безопасности к событиям безопасности
Примеры привязок методов моделей безопасности к событиям безопасности
Модели безопасности KasperskyOS

Прерывание MSI

Прерывание MSI (англ. Message Signaled Interrupt, MSI) – аппаратное прерывание, которое возникает при обращении устройства к контроллеру прерываний через память MMIO.

Связанные разделы:
Управление обработкой прерываний (irq.h)

Приемник уведомлений

Объект ядра KasperskyOS, в котором накапливаются уведомления о событиях, происходящих с ресурсами.

Связанные разделы:
Использование уведомлений (notice_api.h)
Управление дескрипторами (handle_api.h)

Прикладная программа

Программа, которая предназначена для взаимодействия с пользователем решения и выполнения пользовательских задач.

Связанные разделы:
Сборка решения на базе KasperskyOS

Программа

Код, исполняющийся в контексте отдельного процесса.

Связанные разделы:
Сборка решения на базе KasperskyOS

Профиль аудита безопасности

Набор конфигураций аудита безопасности, каждая из которых задает объекты моделей безопасности, покрываемые аудитом безопасности, а также условия выполнения аудита безопасности.

Связанные разделы:
Создание профилей аудита безопасности
Привязка методов моделей безопасности к событиям безопасности
Примеры профилей аудита безопасности
Установка глобальных параметров политики безопасности решения на базе KasperskyOS

Процесс

Запущенная на исполнение программа, которая имеет следующие особенности:

- может предоставлять службы другим процессам и/или использовать службы других процессов через механизм IPC;
- использует службы ядра KasperskyOS через механизм IPC;
- ассоциируется с политикой безопасности решения, которая регулирует взаимодействия процесса с другими процессами и ядром KasperskyOS.

Связанные разделы:
Общие сведения
Запуск процессов
Шаблон init.yaml.in

Прямой доступ к памяти

Прямой доступ к памяти (англ. Direct Memory Access, DMA) – режим обмена данными между устройствами и оперативной памятью, при котором процессор не используется.

Связанные разделы:
Использование DMA (dma.h)
Управление изоляцией памяти для ввода-вывода (iommu_api.h)

Разыменование дескриптора

Операция, при которой клиент отправляет серверу дескриптор, а сервер получает указатель на контекст передачи ресурса, маску прав отправленного дескриптора и предка отправленного клиентом дескриптора, которым сервер уже владеет. Разыменование выполняется, когда клиент, вызывая методы работы с ресурсом (например, чтения, записи, закрытия доступа), передает серверу дескриптор, который был получен от этого сервера при открытии доступа к ресурсу.

Связанные разделы:
Управление дескрипторами (handle_api.h)
Разыменование дескрипторов

Рекурсивный мьютекс

Мьютекс, который может быть захвачен одним потоком исполнения несколько раз.

Связанные разделы:
Использование примитивов синхронизации (event.h, mutex.h, rwlock.h, semaphore.h, condvar.h)

Ресурс

Объект программно-аппаратной системы на базе KasperskyOS, к которому предоставляется доступ процессам. Существуют системные и пользовательские ресурсы.

Связанные разделы:
Управление доступом к ресурсам

Решение модуля безопасности

Решение о разрешении или запрете конкретного взаимодействия процесса с другим процессом или ядром KasperskyOS.

Связанные разделы:
Общие сведения
Управление IPC
Общие сведения об описании политики безопасности решения на базе KasperskyOS

Решение на базе KasperskyOS

Системное ПО (включая ядро KasperskyOS и модуль безопасности Kaspersky Security Module) и прикладное ПО, интегрированные для работы в составе программно-аппаратного комплекса.

Связанные разделы:
Общие сведения
Структура и запуск решения на базе KasperskyOS
Сборка решения на базе KasperskyOS

Семафор

Примитив синхронизации, основанный на счетчике, значение которого может быть атомарно изменено.

Связанные разделы:
Использование примитивов синхронизации (event.h, mutex.h, rwlock.h, semaphore.h, condvar.h)
Ограничения поддержки POSIX

Сервер

В контексте IPC *сервер* – серверный процесс.

Серверная библиотека компонента решения

Транспортная библиотека, которая преобразует IPC-запросы в локальные вызовы.

Связанные разделы:
Транспортный код для IPC

Серверный процесс

Процесс, предоставляющий службы другим процессам через механизм IPC. Один процесс может использовать одновременно несколько IPC-каналов. При этом для одних IPC-каналов процесс может быть сервером, а для других IPC-каналов этот же процесс может быть клиентом.

Связанные разделы:
Механизм IPC

Система паттернов безопасности

Набор паттернов безопасности вместе с инструкциями по их реализации, сочетанию и практическому использованию в проектировании безопасных программных систем.

Связанные разделы:
Паттерны безопасности при разработке под KasperskyOS

Системная программа

Программа, которая создает инфраструктуру для прикладных программ (например, обеспечивает работу с аппаратурой, поддерживает механизм IPC, реализует файловые системы и сетевые протоколы).

Связанные разделы:
Сборка решения на базе KasperskyOS

Системный ресурс

Ресурс, которым управляет ядро KasperskyOS. К системным ресурсам относятся, например, процессы, регионы памяти, прерывания.

Связанные разделы:
Управление доступом к ресурсам
Управление дескрипторами (handle_api.h)

Служба

Набор связанных по смыслу методов, доступных через механизм IPC (например, служба для приема и передачи данных по сети, служба для работы с прерываниями).

Связанные разделы:
Общие сведения
Механизм IPC
Методы служб ядра KasperskyOS

Слушающий дескриптор

Слушающий дескриптор (англ. listener handle) – это серверный IPC-дескриптор, который имеет расширенные права, позволяющие добавлять IPC-каналы в набор идентифицируемых этим дескриптором IPC-каналов.

Связанные разделы:
Создание IPC-каналов
Создание дескрипторов
Инициализация IPC-транспорта для межпроцессного взаимодействия и управление обработкой IPC-запросов (transport-kos.h, transport-kos-dispatch.h)
Динамическое создание IPC-каналов (cm_api.h, ns_api.h)

Событие

Примитив синхронизации, который используется для уведомления одного или нескольких потоков исполнения о выполнении требуемого этим потокам условия.

Связанные разделы:
Использование примитивов синхронизации (event.h, mutex.h, rwlock.h, semaphore.h, condvar.h)

Событие безопасности

Сигнал об инициации взаимодействия процесса с другим процессом или ядром KasperskyOS.

Связанные разделы:
Общие сведения об описании политики безопасности решения на базе KasperskyOS
Привязка методов моделей безопасности к событиям безопасности
Примеры привязок методов моделей безопасности к событиям безопасности

Точка рабочих характеристик

Точка рабочих характеристик (англ. Operating Performance Point, OPP) – комбинация соответствующих друг другу частоты и напряжения для процессорной группы.

Связанные разделы:
Служба управления частотой процессоров

Транспортная библиотека

Чтобы использовать поставляемый компонент решения через IPC, в составе KasperskyOS SDK есть следующие транспортные библиотеки:

- клиентская библиотека компонента решения, которая преобразует локальные вызовы в IPC-запросы;
- серверная библиотека компонента решения, которая преобразует IPC-запросы в локальные вызовы.

Связанные разделы:
Транспортный код для IPC

Транспортный код

Методы и типы на языке C для осуществления IPC.

Связанные разделы:
Транспортный код для IPC
Пример генерации транспортных методов и типов

Транспортный контейнер дескриптора

Структура, состоящая из трех полей: поля дескриптора, поля маски прав дескриптора и поля контекста передачи ресурса. Используется для передачи дескрипторов через IPC.

Связанные разделы:
Передача дескрипторов
Пример использования OCap

Уровень аудита безопасности

Уровень аудита безопасности является глобальным параметром политики безопасности решения и представляет собой беззнаковое целое число, которое задает активную конфигурацию аудита безопасности. (Слово "уровень" здесь означает вариант конфигурации и не предполагает обязательной иерархии.)

Связанные разделы:
Создание профилей аудита безопасности
Установка глобальных параметров политики безопасности решения на базе KasperskyOS

Уровень целостности ресурса

Степень доверия к ресурсу. Степень доверия к ресурсу определяется, например, с учетом того, был ли этот ресурс создан доверенным субъектом внутри программно-аппаратной системы под управлением KasperskyOS или получен из недоверенной внешней программно-аппаратной системы.

Связанные разделы:
Модель безопасности Mic

Уровень целостности субъекта

Степень доверия к субъекту. Степень доверия к субъекту определяется, например, исходя из того, взаимодействует ли субъект с недоверенными внешними программно-аппаратными системами, или имеет ли субъект доказанный уровень качества.

Связанные разделы:
Модель безопасности Mic

Условная переменная

Примитив синхронизации, который используется для уведомления одного или нескольких потоков исполнения о выполнении требуемого этим потокам условия. Используется совместно с мьютексом.

Связанные разделы:
Использование примитивов синхронизации (event.h, mutex.h, rwlock.h, semaphore.h, condvar.h)

Фиксированная часть IPC-сообщения

Часть IPC-сообщения, которая содержит RIID, MID и опционально параметры интерфейсных методов фиксированного размера.

Связанные разделы:
Обзор: структура IPC-сообщения
Работа с ареной IPC-сообщений
Типы данных в языке IDL

Формальная спецификация компонента решения на базе KasperskyOS

Система EDL-, CDL- и IDL-описаний компонента решения (IDL- и CDL-описания опциональны).

Связанные разделы:
Формальные спецификации компонентов решения на базе KasperskyOS

Шаблон безопасности

Шаблон безопасности (также паттерн безопасности) описывает конкретную повторяющуюся проблему безопасности, которая возникает в определенных известных контекстах, а также предлагает хорошо зарекомендовавшую себя общую схему решения такой проблемы безопасности. Шаблон это не законченный проект, который можно преобразовать непосредственно в код, а решение общей проблемы, встречающейся в различных проектах.

Связанные разделы:
Паттерны безопасности при разработке под KasperskyOS

Информация о стороннем коде

Информация о стороннем коде содержится в файле legal_notices.txt, расположенном в папке установки приложения.

Уведомления о товарных знаках

Зарегистрированные товарные знаки и знаки обслуживания являются собственностью их правообладателей.

Arm и Mbed – зарегистрированные товарные знаки или товарные знаки Arm Limited (или дочерних компаний) в США и/или других странах.

Docker и логотип Docker являются товарными знаками или зарегистрированными товарными знаками компании Docker, Inc. в США и/или других странах. Docker, Inc. и другие стороны могут также иметь права на товарные знаки, описанные другими терминами, используемыми в настоящем документе.

Eclipse Mosquitto – товарный знак Eclipse Foundation, Inc.

GoogleTest – товарный знак Google LLC.

Linux – товарный знак Linus Torvalds, зарегистрированный в США и в других странах.

OpenSSL является товарным знаком правообладателя OpenSSL Software Foundation.

Python – товарный знак или зарегистрированный товарный знак Python Software Foundation.

Raspberry Pi – товарный знак Raspberry Pi Foundation.

QT – товарный знак или зарегистрированный товарный знак The Qt Company Ltd.

Ubuntu является зарегистрированным товарным знаком Canonical Ltd.

UNIX – товарный знак, зарегистрированный в США и других странах, использование лицензировано X/Open Company Limited.

Visual Studio, Windows являются товарными знаками группы компаний Microsoft.