kaspersky

KasperskyOS Community Edition 1.1

Содержание

Что нового

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, отладчик GDB, 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 Debian 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:** Debian GNU/Linux® 10 "Buster". 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:

- **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/

- **Arm Mbed TLS (v.2.28.0)** – реализация протоколов TLS и SSL, а также соответствующих криптографических алгоритмов и необходимого кода поддержки.

  Документация: https://github.com/Mbed-TLS/mbedtls

- **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

- **FFmpeg (v.5.1)** – набор библиотек с открытым исходным кодом, которые позволяют записывать, конвертировать и передавать цифровые аудио- и видеозаписи в различных форматах.

  Документация: https://ffmpeg.org/ffmpeg.html

- **fmt (v.8.1.1)** – библиотека для форматирования с открытым исходным кодом.

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

- **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/

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

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

- **libjpeg-turbo (v.2.0.91)** – библиотека для работы с JPEG-изображениями.

  Документация: https://libjpeg-turbo.org/

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

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

- **libpng (v.1.6.38)** – библиотека для работы с PNG-изображениями.

  Документация: http://www.libpng.org/pub/png/libpng.html

- **libxml2 (v.2.9.14)** – библиотека для работы с XML.

  Документация: http://xmlsoft.org/

- **Eclipse Mosquitto (v.2.0.14)** – брокер сообщений, реализующий протокол MQTT.
  Документация: https://mosquitto.org/documentation/
- **nlohmann_json (v.3.9.1)** – библиотека для работы с форматом JSON.
  Документация: https://github.com/nlohmann/json
- **NTP (v.4.2.8P15)** – библиотека для работы протоколом времени NTP.
  Документация: http://www.ntp.org/documentation.html
- **opencv (v.4.6.0)** – библиотека компьютерного зрения с открытым исходным кодом.
  Документация: https://docs.opencv.org/
- **OpenSSL (v.1.1.1q)** – 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/
- **protobuf (v.3.19.4)** – библиотека для сериализации данных.
  Документация: https://developers.google.com/protocol-buffers/docs/overview
- **spdlog (v.1.9.2)** – библиотека для журналирования.
  Документация: https://github.com/gabime/spdlog
- **sqlite (v.3.39.2)** – библиотека для работы с базами данных.
  Документация: https://www.sqlite.org/docs.html
- **Zlib (v.1.2.12)** – 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/
- **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. Не поддерживается динамическая загрузка библиотек.
2. Максимальное поддерживаемое количество запущенных программ: 32.

3. 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.

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

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

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:

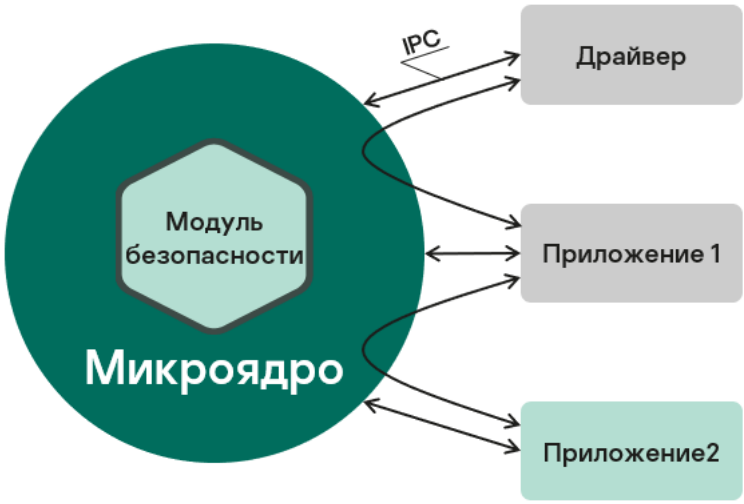
- может предоставлять службы другим процессам и/или использовать службы других процессов через механизм IPC;
- использует службы ядра через механизм IPC;
- ассоциируется с правилами безопасности, которые регулируют взаимодействия процесса с другими процессами и ядром.

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. Чтобы описать политику безопасности решения, используется специально разработанный язык PSL (Policy Specification Language). 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*).

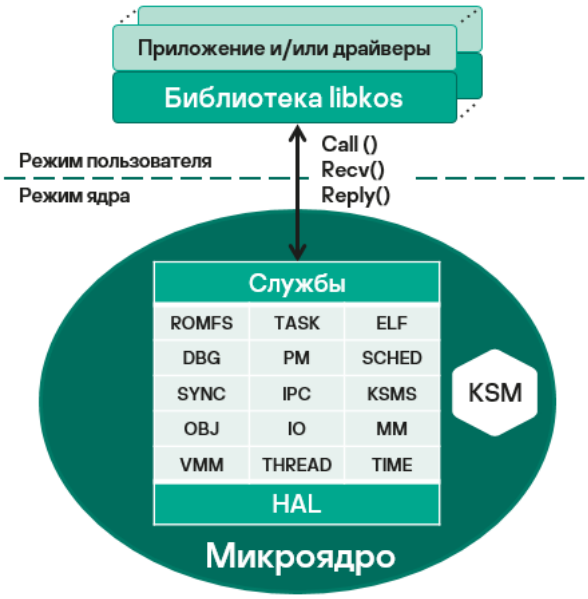
Транспортный код генерируется компилятором `nk-gen-c` из декларативных описаний на языках IDL (Interface Definition Language), CDL (Component Definition Language) и EDL (Entity Definition Language) соответственно (подробнее см. "Формальные спецификации компонентов решения на базе KasperskyOS").

Исходный код модуля безопасности Kaspersky Security Module генерируется компилятором `nk-psl-gen-c` из описания политики безопасности решения и IDL-, CDL-, EDL-описаний.

Исходный код инициализирующей программы генерируется утилитой `einit` из описания инициализации решения (в формате YAML) и IDL-, CDL-, EDL-описаний.

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, DMA-буферов. При наличии у аппаратуры блока управления памятью для операций ввода-вывода (англ. Input-Output Memory Management Unit, IOMMU) подсистема задействуется для более высоких гарантий разделения памяти.
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, фиксирование, освобождение. Работа с таблицами страниц памяти для изоляции адресных пространств процессов.
THREAD	Менеджер потоков	Управление потоками исполнения: создание, уничтожение, приостановка и возобновление.
TIME	Подсистема часов реального времени	Получение и установка системного времени. Использование таймеров, предоставляемых аппаратурой.
SCHED	Планировщик	Поддержка трех классов планирования: потоки реального времени, потоки общего назначения и IDLE – состояние, когда нет потока, готового для исполнения.
SYNC	Подсистема, обеспечивающая примитивы синхронизации	Реализация базовых примитивов синхронизации: спинлок (англ. spinlock), мьютекс (англ. mutex), событие (англ. event). Ядро поддерживает лишь 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: запуск, завершение, приостановка и возобновление. Получение характеристик запущенных процессов (например, имени, пути и приоритета) и кодов их завершения.
ELF	Подсистема загрузки	Загрузка исполняемых ELF-файлов из ROMFS в оперативную память, разбор заголовков ELF-файлов.

	исполняемых 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

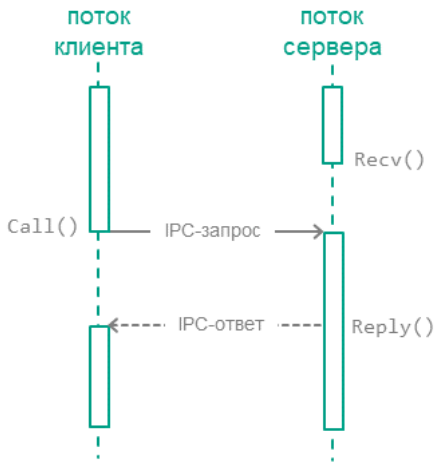
В KasperskyOS процессы взаимодействуют между собой, обмениваясь IPC-сообщениями: *IPC-запросом* и *IPC-ответом*. Во взаимодействии процессов выделяется две роли: *клиент* (процесс, инициирующий взаимодействие) и *сервер* (процесс, обрабатывающий обращение). При этом процесс, являющийся клиентом в одном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

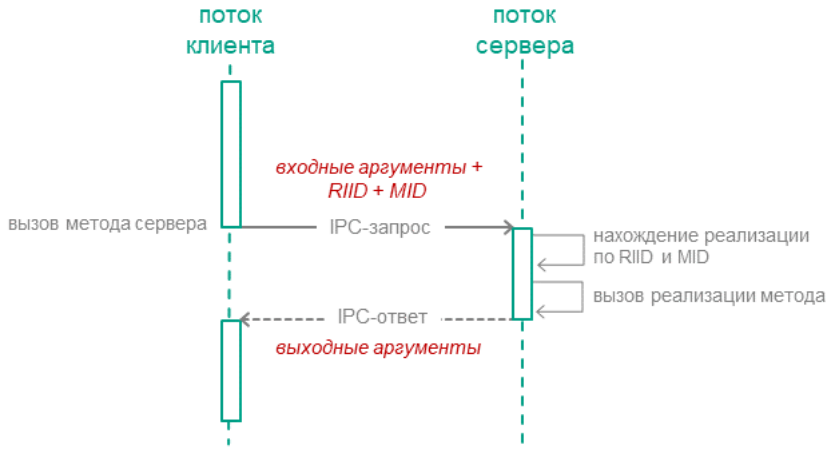
3. При получении IPC-ответа клиентский поток разблокируется и продолжает исполнение.



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 предполагает создание IPC-каналов при запуске решения. Статическое создание IPC-каналов выполняется инициализирующей программой.

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

Управление IPC

Модуль безопасности Kaspersky Security Module интегрирован в механизм, реализующий IPC. Содержимое 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:

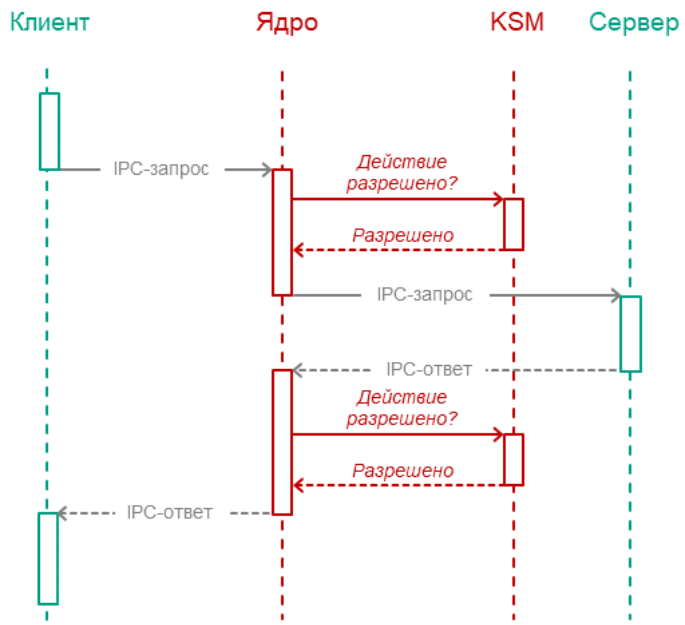
1. Модуль безопасности проверяет, что IPC-сообщение соответствует вызываемому методу службы (проверяются размер IPC-сообщения, а также размер и размещение некоторых структурных элементов).

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 корректное создание IPC-сообщений, их упаковку, отправку и распаковку. 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.

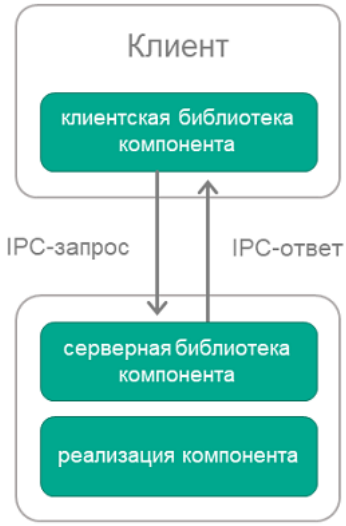
Транспортный код для поставляемых компонентов решения

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 передачи 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 маски 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 `services/ocap.h`). 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 `services/ocap.h`. Структура специальной части для пользовательских ресурсов определяется поставщиком ресурсов с использованием макроса `OCAP_HANDLE_SPEC()`, который определен в заголовочном файле `services/ocap.h`. Поставщику ресурсов необходимо экспортировать публичные заголовочные файлы с описанием структуры специальной части.

При создании дескриптора системного ресурса маска прав задается ядром KasperskyOS, которое применяет маски прав из заголовочного файла `services/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.

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, 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 все остальные программы, входящие в решение.

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 Debian Buster 10.12 из публичного репозитория Docker Hub выполните следующую команду:

   `docker pull debian:10.12`
3. Для запуска образа выполните следующую команду:

   `docker run --net=host --user root --privileged -it --rm debian:10.12 bash`
4. Скопируйте deb-пакет для установки KasperskyOS Community Edition в контейнер.
5. Установите KasperskyOS Community Edition.
6. Для корректной работы некоторых примеров необходимо:
   a. Добавить внутри контейнера директорию `/usr/sbin` в переменную окружения `PATH`, выполнив следующую команду:

      `export PATH=/usr/sbin:$PATH`
   b. Установить программу `parted` внутри контейнера. Для этого добавьте следующую строку в `/etc/apt/sources.list`:

      `deb http://deb.debian.org/debian bullseye main`

      После этого выполните следующую команду:

      `sudo apt update && sudo apt install parted`

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 с root-правами команду:

```
$ apt install <путь-к-deb-пакету>
```

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

Для удобства работы вы можете добавить путь к бинарным файлам инструментов KasperskyOS Community Edition в переменную `PATH`, это позволит работать с утилитами через терминал из любой директории:

```
$ export PATH=$PATH:/opt/KasperskyOS-Community-Edition-<version>/toolchain/bin
```

Удаление

Для удаления KasperskyOS Community Edition выполните с root-правами команду:

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

При этом 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 домашней директории, поэтому директорию с примером, который требуется собрать, нужно скопировать из `/opt/KasperskyOS-Community-Edition-<version>/examples` в домашнюю директорию.

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. 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:

1. Перейдите в директорию с примером.
2. Откройте файл скрипта `cross-build.sh` в текстовом редакторе.
3. В последней строке скрипта замените команду `make sim` на команду `make kos-image`.
4. Сохраните файл скрипта, а затем выполните команду:

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

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

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 некоторых примеров требует использования дополнительных параметров QEMU. Команды для запуска таких примеров приведены в описаниях этих примеров.

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 примеров 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).

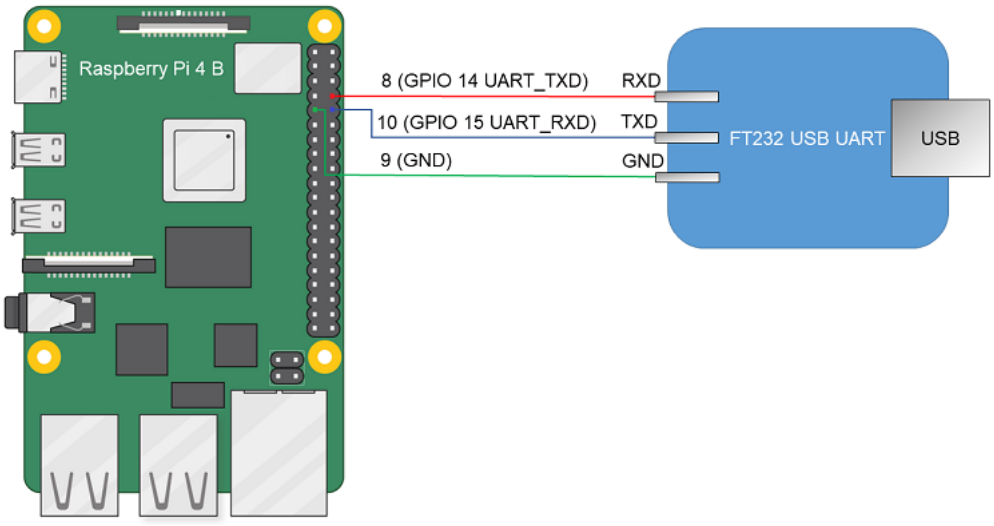


Схема соединения преобразователя USB-UART и Raspberry Pi 4 B

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

3. Установите PuTTY или другую аналогичную программу для чтения данных из COM-порта. Настройте параметры следующим образом: `bps = 115200`, `data bits = 8`, `stop bits = 1`, `parity = none`, `flow control = none`.

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

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, соответствующий ревизии используемой
# Raspberry Pi. Поддерживаются ревизии 1.1, 1.2, 1.4 и 1.5.
# Например, если используется ревизия 1.1, выполните:
$ sudo /opt/KasperskyOS-Community-Edition-
<version>/examples/rpi4_prepare_fs_image_rev1.1.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 dd bs=64k if=path_to_img 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 xz-utils device-tree-compiler
$ git clone https://github.com/u-boot/u-boot.git u-boot-armv8
# Только для Raspberry Pi 4 B ревизий 1.1 и 1.2
$ cd u-boot-armv8 && git checkout tags/v2020.10
# Только для Raspberry Pi 4 B ревизий 1.4 и 1.5
$ cd u-boot-armv8 && git checkout tags/v2022.01
# Для всех ревизий Raspberry Pi
$ make ARCH=arm CROSS_COMPILE=aarch64-linux-gnu- rpi_4_defconfig
# В меню, которое появится при выполнении следующей команды, в разделе
# Boot options замените значение в поле bootcmd value на следующее:
# fatload mmc 0 ${loadaddr} kos-image; bootelf ${loadaddr},
# а в поле preboot default value удалите значение "usb start;".
# Выйдите из меню, сохранив параметры.
$ make ARCH=arm CROSS_COMPILE=aarch64-linux-gnu- menuconfig
$ make ARCH=arm CROSS_COMPILE=aarch64-linux-gnu- u-boot.bin
```

2. Подготовьте образ с файловой системой для SD-карты. Для этого подключите SD-карту к компьютеру и выполните следующие команды:

```
# Только для Raspberry Pi 4 B ревизий 1.1 и 1.2
$ wget https://downloads.raspberrypi.org/raspbian_lite/images/raspbian_lite-2020-
02-14/2020-02-13-raspbian-buster-lite.zip
$ unzip 2020-02-13-raspbian-buster-lite.zip
$ loop_device=$(sudo losetup --find --show --partscan 2020-02-13-raspbian-buster-
lite.img)
# Только для Raspberry Pi 4 B ревизии 1.4
$ wget https://downloads.raspberrypi.org/raspios_lite_arm64/images/raspios_lite_arm6
2022-04-07/2022-04-04-raspios-bullseye-arm64-lite.img.xz
$ unxz 2022-04-04-raspios-bullseye-arm64-lite.img.xz
$ loop_device=$(sudo losetup --find --show --partscan 2022-04-04-raspios-bullseye-
arm64-lite.img)
# Только для Raspberry Pi 4 B ревизии 1.5
$ wget https://downloads.raspberrypi.org/raspios_lite_arm64/images/raspios_lite_arm6
2022-09-07/2022-09-06-raspios-bullseye-arm64-lite.img.xz
$ unxz 2022-09-06-raspios-bullseye-arm64-lite.img.xz
$ loop_device=$(sudo losetup --find --show --partscan 2022-09-06-raspios-bullseye-
arm64-lite.img)
# Для всех ревизий Raspberry Pi
# Образ будет содержать boot-раздел на 1 ГБ в fat32 и три раздела по 256 МБ в ext2,
ext3 и ext4 соответственно.
$ sudo parted ${loop_device} rm 2
$ sudo parted ${loop_device} resizepart 1 1G
$ sudo parted ${loop_device} mkpart primary ext2 1000 1256M
$ sudo parted ${loop_device} mkpart primary ext3 1256 1512M
$ sudo parted ${loop_device} mkpart primary ext4 1512 1768M
$ sudo mkfs.ext2 ${loop_device}p2
$ sudo mkfs.ext3 ${loop_device}p3
$ sudo mkfs.ext4 -O ^64bit,^extent ${loop_device}p4
$ sudo losetup -d ${loop_device}
# В следующей команде [X] – последний символ в имени блочного устройства
# для SD-карты.
$ sudo dd bs=64k if=$(ls *rasp*lite.img) of=/dev/sd[X] conv=fsync
```

3. Скопируйте загрузчик U-Boot на SD-карту, выполнив следующие команды:

```
# В следующих командах путь ~/mnt/fat32 используется для примера. Вы
# можете использовать другой путь.
$ mkdir -p ~/mnt/fat32
# В следующей команде [X] – последний буквенный символ в имени блочного
# устройства для раздела на отформатированной SD-карте.
$ sudo mount /dev/sd[X]1 ~/mnt/fat32/
$ sudo cp u-boot.bin ~/mnt/fat32/u-boot.bin
# Только для Raspberry Pi 4 B ревизии 1.5
# В следующих командах путь ~/tmp_dir используется для примера. Вы
# можете использовать другой путь.
$ mkdir -p ~/tmp_dir
$ cp ~/mnt/fat32/bcm2711-rpi-4-b.dtb ~/tmp_dir
$ dtc -I dtb -O dts -o ~/tmp_dir/bcm2711-rpi-4-b.dts ~/tmp_dir/bcm2711-rpi-4-b.dtb
&& \
$ sed -i -e "0,/emmc2bus = /s/emmc2bus =.*//" ~/tmp_dir/bcm2711-rpi-4-b.dts && \
$ sed -i -e "s/dma-ranges = <0x00 0xc0000000 0x00 0x00 0x40000000>;/dma-ranges =
<0x00 0x00 0x00 0x00 0xfc000000>;/" ~/tmp_dir/bcm2711-rpi-4-b.dts && \
$ sed -i -e "s/mmc@7e340000 {/mmc@7e340000 {\n\t\t\tranges = <0x00 0x7e000000 0x00
0xfe000000 0x1800000>;\n dma-ranges = <0x00 0x00 0x00 0x00 0xfc000000>;/"
~/tmp_dir/bcm2711-rpi-4-b.dts && \
$ dtc -I dts -O dtb -o ~/tmp_dir/bcm2711-rpi-4-b.dtb ~/tmp_dir/bcm2711-rpi-4-b.dts
$ sudo cp ~/tmp_dir/bcm2711-rpi-4-b.dtb ~/mnt/fat32/bcm2711-rpi-4-b.dtb
$ sudo rm -rf ~/tmp_dir
```

4. Заполните конфигурационный файл для загрузчика U-Boot на SD-карте используя следующие команды:

```
$ echo "[all]" > ~/mnt/fat32/config.txt
$ echo "arm_64bit=1" >> ~/mnt/fat32/config.txt
$ echo "enable_uart=1" >> ~/mnt/fat32/config.txt
$ echo "kernel=u-boot.bin" >> ~/mnt/fat32/config.txt
$ echo "dtparam=i2c_arm=on" >> ~/mnt/fat32/config.txt
$ echo "dtparam=i2c=on" >> ~/mnt/fat32/config.txt
$ echo "dtparam=spi=on" >> ~/mnt/fat32/config.txt
$ sync
$ sudo umount ~/mnt/fat32
```

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. Скопируйте на загрузочную SD-карту образ решения на базе KasperskyOS. Для этого подключите загрузочную SD-карту к компьютеру и выполните следующие команды:

```
# В следующей команде [X] – последний буквенный символ в имени блочного
# устройства для раздела на загрузочной SD-карте.
# В следующих командах путь ~/mnt/fat32 используется для примера. Вы
```

```
# можете использовать другой путь.
$ sudo mount /dev/sd[X]1 ~/mnt/fat32/
$ sudo cp build/einit/kos-image ~/mnt/fat32/kos-image
$ sync
$ sudo umount ~/mnt/fat32
```

4. Подключите загрузочную SD-карту к Raspberry Pi 4 B.

5. Подайте питание на Raspberry Pi 4 B и дождитесь, пока запустится пример.

   О том, что пример запустился, свидетельствует вывод, отображаемый на компьютере.

Запуск процессов

Обзор: Einit и init.yaml

Инициализирующая программа Einit

При старте ядро KasperskyOS находит в образе решения и запускает исполняемый файл с именем `Einit` (*инициализирующая программа*). Запущенный процесс имеет класс `Einit` и, как правило, используется для запуска остальных процессов, которые требуются в момент старта решения.

Генерация C-кода инициализирующей программы

В составе пакета инструментов KasperskyOS Community Edition поставляется утилита `einit`, которая позволяет сгенерировать C-код инициализирующей программы на основе *init-описания* (файл с описанием обычно имеет имя `init.yaml`). Полученная программа использует KasperskyOS API для выполнения следующих действий:

- статическое создание и запуск процессов;
- статическое создание IPC-каналов.

Стандартным способом использования утилиты `einit` является интеграция ее вызова в один из шагов сборочного скрипта, в результате которого утилита `einit` на основе файла `init.yaml` сгенерирует файл `einit.c`, содержащий 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 статических описаний. Эти файлы поставляются в составе пакета инструментов KasperskyOS Community Edition и автоматически подключаются при сборке решения. Однако класс процессов `Einit` должен быть описан в файле `security.psl`.

Синтаксис init.yaml

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

- процессы, запускаемые при загрузке KasperskyOS;
- IPC-каналы, используемые процессами для взаимодействия между собой.

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	Класс безопасности процесса
`task`	Нет	Имя процесса. Если его не указывать, то будет взято имя класса безопасности. У каждого процесса должно быть уникальное имя. Можно запустить несколько процессов одного класса безопасности, но с разными именами.
`path`	Нет	Имя исполняемого файла в ROMFS (в образе решения), из которого будет запущен процесс. Если его не указывать, то будет взято имя класса безопасности без "префиксов" и точек. Например, процессы классов безопасности `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 дескрипторами которых будет владеть процесс. По умолчанию список пуст. (Помимо статически создаваемых IPC-каналов процессы могут использовать динамически создаваемые IPC-каналы.)
`args`	Нет	Список аргументов, передаваемых процессу (функции `main()`). Максимальный размер одного элемента списка – **1024** байта.
`env`	Нет	Словарь переменных окружения, передаваемых процессу. Ключами в этом словаре являются имена переменных, которым сопоставлены передаваемые значения. Максимальный размер значения – **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 собраны init-описания, демонстрирующие различные аспекты запуска процессов.

> В примерах в составе KasperskyOS Community Edition может использоваться формат init-описания с макросами (`init.yaml.in`).

Файл с init-описанием обычно называется `init.yaml`, хотя может иметь любое имя.

Соединение и запуск процесса-клиента и процесса-сервера

В следующем примере будут запущены два процесса: класса `Client` и 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. Процессы будут соединены IPC-каналом с именем `server_connection`.

init.yaml

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

Указание исполняемого файла для запуска

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

init.yaml

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

Запуск двух процессов из одного исполняемого файла

В следующем примере будут запущены три процесса: процесс класса `Client` из исполняемого файла по умолчанию (`Client`), а также процессы классов `MainServer` и `BkServer` из исполняемого файла `srv`. Имена процессов не указаны, поэтому они будут совпадать с именами классов процессов.

init.yaml

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

Запуск двух процессов одного класса

В следующем примере будут запущены: один процесс класса `Client` (с именем по умолчанию – `Client`) и два процесса класса `Server` с именами `UserServer` и `PrivilegedServer`. Клиентский процесс связан с серверными процессами IPC-каналами с именами `server_connection_us` и `server_connection_ps`, соответственно. 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
```

Передача переменных окружения и аргументов функции main()

В следующем примере будут запущены два процесса: один класса `VfsFirst` (с именем по умолчанию – `VfsFirst`) и второй класса `VfsSecond` (с именем по умолчанию – `VfsSecond`). Первый процесс при запуске получит аргумент `-f /etc/fstab`, а также переменные окружения: `ROOTFS` со значением **ramdisk0,0 / ext2 0** и `UNMAP_ROMFS` со значением **1**. Второй процесс при запуске получит аргумент `-l devfs /dev devfs 0`.

Имена исполняемых файлов не указаны, поэтому они будут совпадать с именами классов процессов.

> Если в решении используется программа Env, то передаваемые через нее аргументы и переменные окружения переопределяют значения, заданные через `init.yaml`.

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 процесса с помощью KasperskyOS API

> **В этом примере:** использование функций `EntityInitEx()` и `EntityRun()` для запуска исполняемого файла из образа решения.

Ниже приводится код функции `GpMgrOpenSession()`, выполняющей запуск серверного процесса, соединение его с клиентским процессом и инициализацию IPC-транспорта. Исполняемый файл нового процесса должен содержаться в ROMFS-хранилище решения.

```
#define CONNECT_RETRY 150   /* Количество попыток соединения */
#define CONNECT_DELAY 10    /* Задержка в мс между попытками */

/**
 * Параметр classname задает имя класса запускаемого процесса,
 * параметр server задает уникальное имя процесса, а параметр service содержит имя
сервиса,
 * используемое при динамическом создании канала.
 * Выходной параметр transport содержит инициализированный транспорт,
 * если IPC-канал до клиента успешно создан.
 */
Retcode GpMgrOpenSession(const char *classname, const char *server,
                const char *service, NkKosTransport *transport)
{
    Retcode rc;
    Entity *e;
    EntityInfo tae_info;
    Handle endpoint;
    rtl_uint32_t riid;
    int count = CONNECT_RETRY;

    /* Инициализация структуры описания процесса. */
    rtl_memset(&tae_info, 0, sizeof(tae_info));
    tae_info.eiid = classname;
    tae_info.args[0] = server;
    tae_info.args[1] = service;

    /* Создание процесса с описанием tae_info и именем server.
     * Поскольку третий параметр равен RTL_NULL, имя запускаемого
     * бинарного файла совпадает с именем класса из описания tae_info.
     * Созданный процесс находится в остановленном состоянии. */
    if ((e = EntityInitEx(&tae_info, server, RTL_NULL)) == NK_NULL)
    {
        rtl_printf("Cannot init entity '%s'\n", tae_info.eiid);
        return rcFail;
    }

    /* Запуск процесса. */
    if ((rc = EntityRun(e)) != rcOk)
    {
        rtl_printf("Cannot launch entity %" RTL_PRId32 "\n", rc);
        EntityFree(e);
        return rc;
    }

    /* Динамическое создание IPC-канала. */
    while ((rc = KnCmConnect(server, service, INFINITE_TIMEOUT, &endpoint, &riid) ==
            rcResourceNotFound && count--)
    {
        KnSleep(CONNECT_DELAY);
    }
```

```
    if (rc != rcOk)
    {
        rtl_printf("Cannot connect to server %" RTL_PRId32 "\n", rc);
        return rc;
    }

    /* Инициализация IPC-транспорта. */
    NkKosTransport_Init(transport, endpoint, NK_NULL, 0);
    ...

    return rcOk;
}
```

> Чтобы процесс мог запускать другие процессы, политика безопасности решения должна разрешать ему использование следующих служб ядра: `Handle`, `Task` и `VMM` (их описания находятся в директории `kl\core\`).

Обзор: программа Env

Служебная программа `Env` предназначена для передачи аргументов и переменных окружения запускаемым процессам. При запуске каждый процесс автоматически отправляет запрос процессу `Env` и получает необходимые данные.

> Обращение процесса к `Env` переопределяет аргументы и переменные окружения, полученные через `Einit`.

Чтобы использовать программу `Env` в своем решении, необходимо:

1. Разработать код программы `Env`, используя макросы из `env/env.h`.

2. Собрать бинарный файл программы `Env`, скомпоновав ее с библиотекой `env_server`.

3. В init-описании указать, что необходимо запустить процесс `Env` и соединить с ней выбранные процессы (`Env` при этом является сервером). Имя канала задается макросом `ENV_SERVICE_NAME`, объявленным в файле `env/env.h`.

4. Включить бинарный файл `Env` в образ решения.

Код программы Env

В коде программы `Env` используются следующие макросы и функции, объявленные в файле `env/env.h`:

- `ENV_REGISTER_ARGS(name,argarr)` – передать процессу с именем `name` аргументы из массива `argarr` (максимальный размер одного элемента – **256** байтов);
- `ENV_REGISTER_VARS(name,envarr)` – передать процессу с именем `name` переменные окружения из массива `envarr` (максимальный размер одного элемента – **256** байтов);

- `ENV_REGISTER_PROGRAM_ENVIRONMENT(name,argarr,envarr)` – передать процессу с именем `name` как аргументы, так и переменные окружения;
- `envServerRun()` – инициализировать серверную часть программы `Env`, чтобы она могла отвечать на запросы.

Примеры использования Env

Передача переменных окружения и аргументов с помощью Env

Пример передачи аргументов при запуске процесса

Ниже приводится код программы `Env`, которая при запуске процесса с именем `NetVfs` передаст ему три аргумента: `NetVfs`, `-l devfs /dev devfs 0` и `-l romfs /etc romfs 0`:

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 0"
    };
    ENV_REGISTER_ARGS("NetVfs", NetVfsArgs);

    envServerRun();
    return EXIT_SUCCESS;
}
```

Пример передачи переменных окружения при запуске процесса

Ниже приводится код программы `Env`, которая при запуске процесса с именем `Vfs3` передаст ему две переменных окружения: `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

Состав компонента VFS

Компонент VFS содержит набор исполняемых файлов, библиотек и файлов описаний, позволяющих использовать файловые системы и/или сетевой стек, вынесенные в отдельный процесс VFS (*Virtual File System – виртуальная файловая система*). При необходимости можно собрать собственные реализации VFS.

Библиотеки VFS

CMake-пакет `vfs` содержит следующие библиотеки:

- `vfs_fs` – содержит реализации defvs, ramfs и romfs, а также позволяет добавить в VFS реализации других файловых систем;
- `vfs_net` – содержит реализацию defvs и сетевого стека;
- `vfs_imp` – содержит в себе сумму компонентов `vfs_fs` и `vfs_net`;
- `vfs_remote` – клиентская транспортная библиотека; преобразует локальные вызовы в IPC-запросы к VFS и принимает IPC-ответы;
- `vfs_server` – серверная транспортная библиотека VFS; принимает IPC-запросы, преобразует их в локальные вызовы и отправляет IPC-ответы;
- `vfs_local` – используется для статической компоновки клиента с библиотеками VFS.

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`, `vfs_imp` и `dnet_imp`.

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

При необходимости можно самостоятельно собрать исполняемый файл VFS с нужной функциональностью.

Файлы описаний VFS

В директории `/opt/KasperskyOS-Community-Edition-<version>/sysroot-aarch64-kos/include/kl/` находятся следующие файлы VFS:

- `VfsRamFs.edl`, `VfsSdCardFs.edl`, `VfsNet.edl` и `VfsEntity.edl` и сгенерированные из них заголовочные файлы с транспортным кодом;
- `Vfs.cdl` и сгенерированный `Vfs.cdl.h`;
- `Vfs*.idl` и сгенерированные из них заголовочные файлы с транспортным кодом.

Создание IPC-канала до VFS

Рассмотрим программу `Client`, использующую файловые системы и сокеты Беркли. Для обработки ее вызовов запустим один процесс VFS (с именем `VfsFsnet`). В этот процесс будут направляться как "сетевые", так и "файловые" вызовы. Такой подход используется в тех случаях, когда не требуется разделение "файловых" и "сетевых" информационных потоков.

Чтобы взаимодействие процессов `Client` и `VfsFsnet` было корректным, имя IPC-канала между ними должно задаваться макросом `_VFS_CONNECTION_ID`, объявленным в файле `vfs/defs.h`.

Ниже приводится фрагмент init-описания для соединения процессов `Client` и `VfsFsnet`.

init.yaml

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

- name: VfsFsnet
```

Сборка исполняемого файла VFS

При сборке исполняемого файла VFS можно включить в него именно ту функциональность, которая требуется, например:

- реализацию той или иной файловый системы;
- сетевой стек;
- сетевой драйвер.

Например, при разделении файловых и сетевых вызовов понадобится сборка "файловой версии" и "сетевой версии" VFS. В некоторых случаях в VFS потребуется включить и сетевой стек, и файловые системы ("полная версия" VFS).

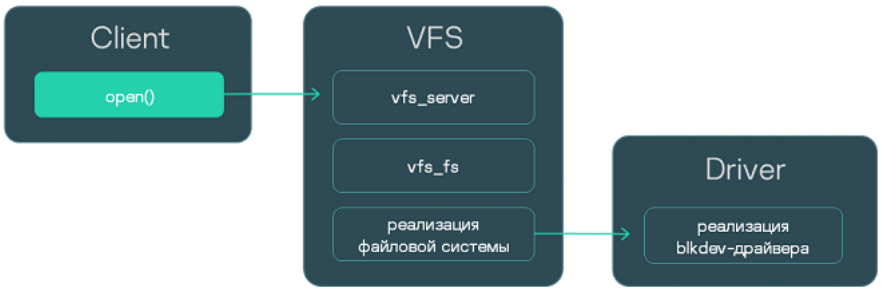
Сборка "файловой версии" VFS

Рассмотрим программу VFS, содержащую только реализацию файловой системы lwext4 и не содержащую сетевого стека. Для сборки такого исполняемого файла необходимо файл с функцией `main()` скомпоновать с библиотеками `vfs_server`, `vfs_fs` и `lwext4`:

CMakeLists.txt

```
project (vfsfs)

include (platform/nk)

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

add_executable (VfsFs "src/vfs.c")

# Компоновка с библиотеками VFS
target_link_libraries (VfsFs
                       ${vfs_SERVER_LIB}
                       ${LWEXT4_LIB}
                       ${vfs_FS_LIB})

# Подготовка VFS для соединения с процессом ramdisk-драйвера
set_target_properties (VfsFs PROPERTIES ${blkdev_ENTITY}_REPLACEMENT
${ramdisk_ENTITY})
```

> Драйвер блочного устройства не может быть скомпонован с VFS, поэтому всегда должен быть запущен как отдельный процесс.



Взаимодействие трех процессов: клиента, "файловой версии" VFS и драйвера блочного устройства

Сборка "сетевой версии" VFS совместно с сетевым драйвером

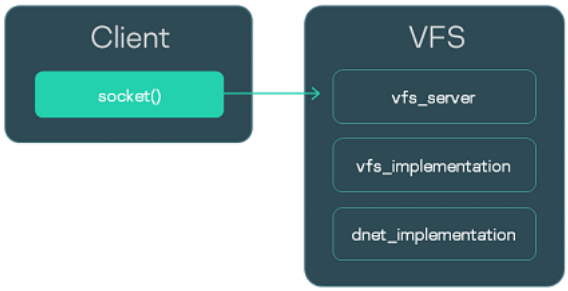
Рассмотрим программу VFS, содержащую сетевой стек с драйвером и не содержащую реализаций файловых систем. Для сборки такого исполняемого файла необходимо файл с функцией `main()` скомпоновать с библиотеками `vfs_server`, `vfs_implementation` и `dnet_implementation`.

CMakeLists.txt

```
project (vfsnet)
```

```
include (platform/nk)

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

add_executable (VfsNet "src/vfs.c")

# Компоновка с библиотеками VFS
target_link_libraries (VfsNet
                       ${vfs_SERVER_LIB}
                       ${vfs_IMPLEMENTATION_LIB}
                       ${dnet_IMPLEMENTATION_LIB})

# Отключение драйвера блочного устройства
set_target_properties (VfsNet PROPERTIES ${blkdev_ENTITY}_REPLACEMENT "")
```

> Библиотека `dnet_implementation` уже включает в себя сетевой драйвер, поэтому запуск отдельного процесса драйвера не требуется.



Взаимодействие процесса Client с процессом "сетевой версии" VFS

Сборка "сетевой версии" VFS с отдельным сетевым драйвером

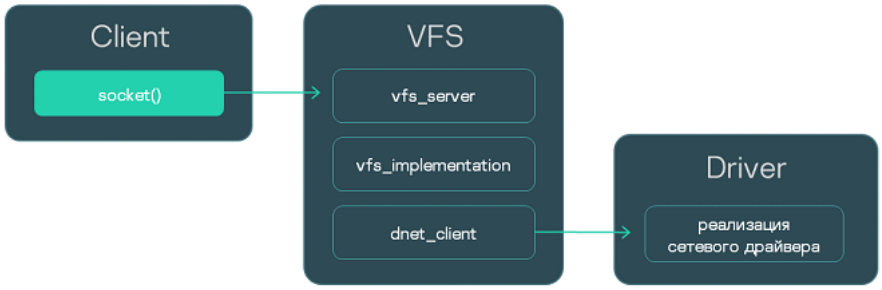
Еще один вариант сборки "сетевой версии" VFS – без сетевого драйвера. Сам сетевой драйвер необходимо будет запустить как отдельный процесс. Взаимодействие с драйвером происходит через IPC с помощью библиотеки `dnet_client`.

Таким образом, необходимо файл с функцией `main()` скомпоновать с библиотеками `vfs_server`, `vfs_implementation` и `dnet_client`.

CMakeLists.txt

```
project (vfsnet)

include (platform/nk)

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

add_executable (VfsNet "src/vfs.c")

# Компоновка с библиотеками VFS
target_link_libraries (VfsNet
```

```
                    ${vfs_SERVER_LIB}
                    ${vfs_IMPLEMENTATION_LIB}
                    ${dnet_CLIENT_LIB})

# Отключение драйвера блочного устройства
set_target_properties (VfsNet PROPERTIES ${blkdev_ENTITY}_REPLACEMENT "")
```



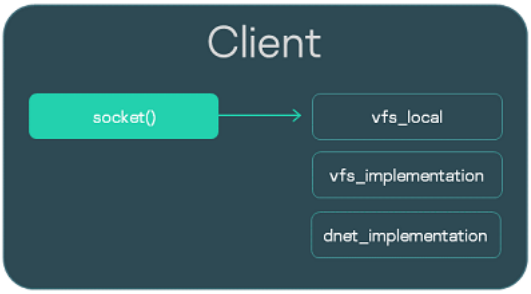
Взаимодействие трех процессов: клиента, "сетевой версии" VFS и сетевого драйвера

Сборка "полной версии" VFS

Если в VFS требуется включить как сетевой стек, так и реализации файловых систем, то при сборке следует использовать библиотеки `vfs_server`, `vfs_implementation`, `dnet_implementation` (или `dnet_client` – в случае отдельного сетевого драйвера), а также библиотеки реализации файловых систем.

Объединение клиента и VFS в один исполняемый файл

Рассмотрим программу `Client`, использующую сокеты Беркли. Вызовы, которые выполняет `Client`, должны направляться в VFS. Обычный путь состоит в запуске отдельного процесса VFS и создании IPC-канала. Однако вместо этого можно интегрировать функциональность VFS в сам исполняемый файл `Client`. Для этого нужно при сборке исполняемого файла `Client` скомпоновать его с библиотекой `vfs_local`, которая будет принимать вызовы, а также с библиотеками реализации – `vfs_implementation` и `dnet_implementation`.



Локальную компоновку с VFS удобно использовать при отладке. Кроме того, вызовы для работы с сетью могут обрабатываться намного быстрее за счет исключения IPC-вызовов. Тем не менее, изоляция VFS в отдельном процессе и IPC-взаимодействие с ним рекомендуется во всех случаях как более безопасный подход.

Ниже приведен сборочный скрипт исполняемого файла `Client`.

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 `Client` использует файловые системы, то помимо `vfs_local`, его нужно скомпоновать с библиотекой `vfs_fs` и реализацией используемой файловой системы. Кроме того, нужно добавить в решение драйвер блочного устройства.

Обзор: аргументы и переменные окружения VFS

Аргументы VFS

- `-l <запись в формате fstab>`

  Аргумент `-l` позволяет монтировать файловую систему.

- `-f <путь к файлу fstab>`

  Аргумент `-f` позволяет передать файл с записями в формате fstab для монтирования файловых систем. Файл будет искаться в ROMFS-хранилище. Если переменная `UMNAP_ROMFS` определена, то файл будет искаться на файловой системе, смонтированной с помощью переменной `ROOTFS`.

Пример использования аргументов -l и -f

Переменные окружения VFS

- `UNMAP_ROMFS`

  Если переменная `UNMAP_ROMFS` определена, то ROMFS-хранилище будет удалено. Это позволяет сэкономить память и изменить поведение при использовании аргумента `-f`.

- `ROOTFS = <запись в формате fstab>`

  Переменная `ROOTFS` позволяет монтировать файловую систему в корневой каталог. В комбинации с переменной `UNMAP_ROMFS` и аргументом `-f` позволяет искать fstab-файл на монтированной файловой системе, а не в ROMFS-хранилище. Пример использования ROOTFS

- `VFS_CLIENT_MAX_THREADS`

  Переменная окружения `VFS_CLIENT_MAX_THREADS` позволяет в момент запуска VFS переопределить параметр конфигурирования SDK `VFS_CLIENT_MAX_THREADS`.

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

Переменная `_VFS_NETWORK_BACKEND` задает используемый для "сетевых" вызовов бэкенд. Можно указать имя стандартного бэкенда: **client**, **server** или **local**, а также имя пользовательского бэкенда. Если используется бэкенд **local**, то имя IPC-канала не указывается (`_VFS_NETWORK_BACKEND=local:`). Может быть указано два и больше IPC-канала через запятую.

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

  Переменная `_VFS_FILESYSTEM_BACKEND` задает используемый для "файловых" вызовов бэкенд. Имя бэкенда и имя IPC-канала до VFS задаются так же, как и для переменной `_VFS_NETWORK_BACKEND`.

Значения по умолчанию

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 файловой системы при старте

При запуске процесса 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 `mount()` после запуска VFS, но и непосредственно при запуске процесса VFS – передав ему нужные аргументы и переменные окружения.

Рассмотрим три примера монтирования файловых систем при запуске VFS. Для передачи аргументов и переменных окружения процессу VFS использована программа `Env`.

Монтирование с помощью аргумента -l

Простой способ монтировать файловую систему – это передать процессу VFS аргумент `-l <запись в формате fstab>`.

В этом примере при запуске процесса с именем `Vfs1` будут монтированы файловые системы devfs и romfs.

env.c

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

int main(int argc, char** argv)
{
    const char* Vfs1Args[] = {
        "-l", "devfs /dev devfs 0",
        "-l", "romfs /etc romfs 0"
    };

    ENV_REGISTER_ARGS("Vfs1", Vfs1Args);

    envServerRun();

    return EXIT_SUCCESS;
}
```

Монтирование с помощью fstab из ROMFS

Если при сборке решения добавить fstab-файл, после старта он будет доступен через ROMFS-хранилище. Его можно использовать для монтирования, передав процессу VFS аргумент `-f <путь к fstab-файлу>`.

В этом примере при запуске процесса с именем `Vfs2` будут монтированы файловые системы, заданные через файл `fstab`, который был добавлен при сборке решения.

env.c

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

int main(int argc, char** argv)
{
    const char* Vfs2Args[] = { "-f", "fstab" };

    ENV_REGISTER_ARGS("Vfs2", Vfs2Args);

    envServerRun();

    return EXIT_SUCCESS;
}
```

Монтирование с помощью "внешнего" fstab

Пусть fstab-файл находится не в ROMFS-образе решения, а на диске. Чтобы использовать его для монтирования, необходимо передать VFS следующие аргументы и переменные окружения:

1. `ROOTFS`. Эта переменная позволяет монтировать в корневую директорию файловую систему, в которой находится fstab-файл.

2. `UNMAP_ROMFS`. Если эта переменная определена, ROMFS-хранилище удаляется. В итоге fstab-файл будет искаться на файловой системе, смонтированной с помощью переменной `ROOTFS`.

3. `-f`. Этот аргумент используется, чтобы задать путь к fstab-файлу.

В следующем примере при запуске процесса с именем `Vfs3` в корневой каталог будет монтирована файловая система ext2, на которой будет найден файл `/etc/fstab` для монтирования дополнительных файловых систем. ROMFS-хранилище будет удалено.

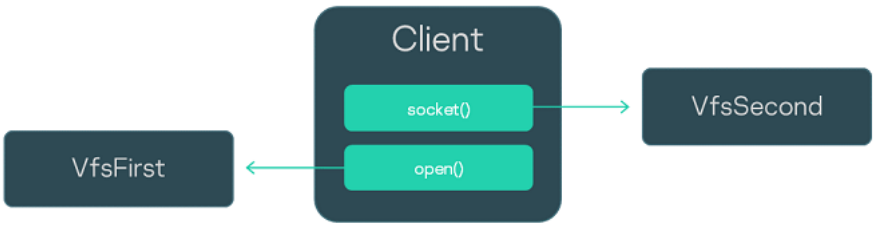
env.c

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

int main(int argc, char** argv)
{
    const char* Vfs3Args[] = { "-f", "/etc/fstab" };

    const char* Vfs3Envs[] = {
        "ROOTFS=ramdisk0,0 / ext2 0",
        "UNMAP_ROMFS=1"
    };

    ENV_REGISTER_PROGRAM_ENVIRONMENT("Vfs3", Vfs3Args, Vfs3Envs);

    envServerRun();

    return EXIT_SUCCESS;
}
```

Разделение файловых и сетевых вызовов с помощью бэкендов VFS

> **В этом примере:** паттерн безопасной разработки, предусматривающий разделение "сетевых" и "файловых" информационных потоков.

Рассмотрим программу `Client`, использующую файловые системы и сокеты Беркли. Для обработки ее вызовов мы запустим не один, а два отдельных VFS-процесса из исполняемых файлов `VfsFirst` и `VfsSecond`. Через переменные окружения мы зададим файловые бэкенды как работающие через канал до `VfsFirst`, а сетевые бэкенды – как работающие через канал до `VfsSecond`. Будем использовать стандартные бэкенды **client** и **server**. Благодаря этому мы перенаправим "файловые вызовы" `Client` в `VfsFirst`, а "сетевые" – в `VfsSecond`. Чтобы передать процессам переменные окружения, добавим в решение программу `Env`.



Init-описание решения представлено ниже. Процесс `Client` будет соединен с процессами `VfsFirst` и `VfsSecond`, при этом каждый из трех процессов соединен с процессом `Env`. Обратите внимание, что имя IPC-канала до процесса `Env` задается с помощью переменной `ENV_SERVICE_NAME`.

`init.yaml`

```
entities:

- name: Env

- name: Client
  connections:
  - target: Env
    id: {var: ENV_SERVICE_NAME, include: env/env.h}
  - target: VfsFirst
    id: VFS1
  - target: VfsSecond
    id: VFS2

- name: VfsFirst
  connections:
  - target: Env
    id: {var: ENV_SERVICE_NAME, include: env/env.h}

- name: VfsSecond
  connections:
  - target: Env
    id: {var: ENV_SERVICE_NAME, include: env/env.h}
```

Чтобы направить все "файловые" вызовы в `VfsFirst`, зададим значение переменной окружения `_VFS_FILESYSTEM_BACKEND` следующим образом:

- для `VfsFirst`: `_VFS_FILESYSTEM_BACKEND=server:<имя IPC-канала до VfsFirst>`;
- для `Client`: `_VFS_FILESYSTEM_BACKEND=client:<имя IPC-канала до VfsFirst>`.

Для направления "сетевых" вызовов в `VfsSecond` используем аналогичную переменную окружения `_VFS_NETWORK_BACKEND`:

- для `VfsSecond` зададим: `_VFS_NETWORK_BACKEND=server:<имя IPC-канала до VfsSecond>`;
- для `Client`: `_VFS_NETWORK_BACKEND=client:<имя IPC-канала до VfsSecond>`.

Значение переменных окружения зададим через программу `Env`, код которой представлен ниже.

`env.c`

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

int main(void)
{
    const char* vfs_first_envs[] = { "_VFS_FILESYSTEM_BACKEND=server:VFS1" };
    ENV_REGISTER_VARS("VfsFirst", vfs_first_envs);

    const char* vfs_second_envs[] = { "_VFS_NETWORK_BACKEND=server:VFS2" };
    ENV_REGISTER_VARS("VfsSecond", vfs_second_envs);

    const char* client_envs[] = { "_VFS_FILESYSTEM_BACKEND=client:VFS1",
"_VFS_NETWORK_BACKEND=client:VFS2" };
    ENV_REGISTER_VARS("Client", client_envs);

    envServerRun();

    return EXIT_SUCCESS;
}
```

Написание пользовательского бэкенда VFS

> **В этом примере:** изменение логики обработки файловых вызовов с помощью специального бэкенда VFS.

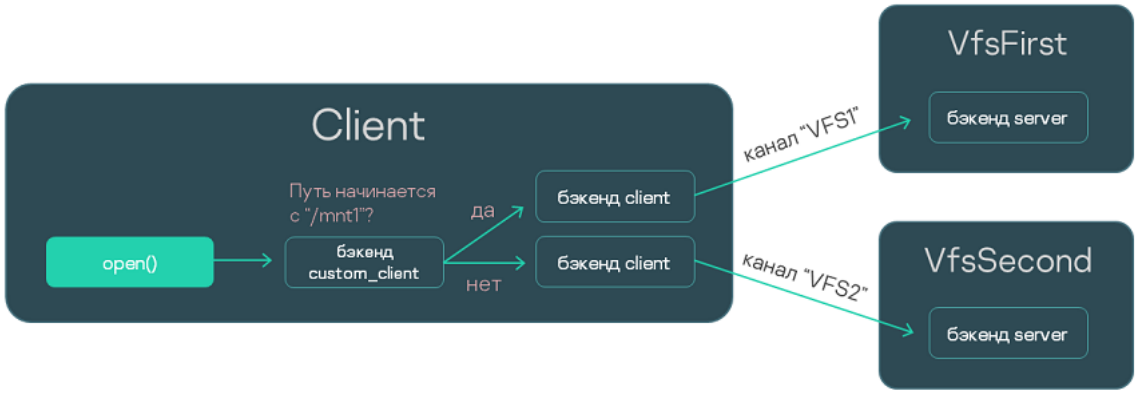
Рассмотрим решение, включающее процессы `Client`, `VfsFirst` и `VfsSecond`. Пусть процесс `Client` соединен с `VfsFirst` и `VfsSecond` с помощью IPC-каналов.

Задача: сделать так, чтобы обращения процесса `Client` к файловой системе fat32 обрабатывались процессом `VfsFirst`, а обращения к ext4 обрабатывались `VfsSecond`. Для решения этой задачи можно использовать механизм бэкендов VFS, причем даже не придется менять код программы `Client`.

Мы напишем пользовательский бэкенд `custom_client`, который будет отправлять вызовы по каналу `VFS1` или `VFS2`, в зависимости от того, начинается ли путь к файлу с **/mnt1**. Для отправки вызовов `custom_client` будет использовать стандартные бэкенды `client`, то есть будет являться проксирующим бэкендом.

С помощью аргумента **-l** мы монтируем fat32 в директорию **/mnt1** для процесса `VfsFirst` и ext4 в **/mnt2** для процесса `VfsSecond`. (Предполагается, что `VfsFirst` содержит реализацию fat32, а `VfsSecond` – реализацию ext4.) С помощью переменной окружения `_VFS_FILESYSTEM_BACKEND` зададим используемые процессами бэкенды (**custom_client** и **server**) и IPC-каналы (**VFS1** и **VFS2**).

Наконец, с помощью init-описания зададим имена IPC-каналов: **VFS1** и **VFS2**.



Ниже мы рассмотрим подробнее:

1. Код бэкенда `custom_client`.
2. Компоновка программы `Client` и бэкенда `custom_client`.
3. Код программы `Env`.
4. Init-описание.

Написание бэкенда custom_client

Этот файл содержит реализацию проксирующего пользовательского бэкенда, передающего вызовы в один из двух стандартных бэкендов **client**. Логика выбора бэкенда зависит от используемого пути или от дескриптора файла и управляется дополнительными структурами данных.

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 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 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 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 initilize 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 initilize client backend!");
    assert(ctx.vfs_ext4->dtor != NULL && "VFS FS backend has not set the
destructor!");

    return &ctx.wrapper;
}

/* Регистрация пользовательского 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 и бэкенда custom_client

Написанный бэкенд скомпилируем в библиотеку:

CMakeLists.txt

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

Готовую библиотеку `backend_client` скомпонуем с программой `Client`:

CMakeLists.txt (фрагмент)

```
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
)
```

Написание программы Env

Чтобы передать процессам аргументы и переменные окружения, используем программу Env.

env.c

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

int main(int argc, char** argv)
{
    /* Монтирование fat32 в /mnt1 для процесса VfsFirst и ext4 в /mnt2 для процесса
VfsSecond. */

    const char* VfsFirstArgs[] = {
        "-l", "ahci0 /mnt1 fat32 0"
    };

    ENV_REGISTER_ARGS("VfsFirst", VfsFirstArgs);

    const char* VfsSecondArgs[] = {
        "-l", "ahci1 /mnt2 ext4 0"
    };

    ENV_REGISTER_ARGS("VfsSecond", VfsSecondArgs);

    /* Задание файловых бэкендов. */

    const char* vfs_first_args[] = { "_VFS_FILESYSTEM_BACKEND=server:VFS1" };
    ENV_REGISTER_VARS("VfsFirst", vfs_first_args);

    const char* vfs_second_args[] = { "_VFS_FILESYSTEM_BACKEND=server:VFS2" };
    ENV_REGISTER_VARS("VfsSecond", vfs_second_args);

    const char* client_fs_envs[] = { "_VFS_FILESYSTEM_BACKEND=custom_client:VFS1,VFS2"
};
    ENV_REGISTER_VARS("Client", client_fs_envs);

    envServerRun();
```

```
    return EXIT_SUCCESS;
}
```

Изменение init.yaml

Для IPC-каналов, соединяющих процесс `Client` с процессами `VfsFirst` и `VfsSecond`, необходимо задать те же имена, которые мы указали в переменной окружения `_VFS_FILESYSTEM_BACKEND`: **VFS1** и **VFS2**.

init.yaml

```
entities:

- name: vfs_backend.Env

- name: vfs_backend.Client
  connections:
  - target: vfs_backend.Env
    id: {var: ENV_SERVICE_NAME, include: env/env.h}
  - target: vfs_backend.VfsFirst
    id: VFS1
  - target: vfs_backend.VfsSecond
    id: VFS2

- name: vfs_backend.VfsFirst
  connections:
  - target: vfs_backend.Env
    id: {var: ENV_SERVICE_NAME, include: env/env.h}

- name: vfs_backend.VfsSecond
  connections:
  - target: vfs_backend.Env
    id: {var: ENV_SERVICE_NAME, include: env/env.h}
```

IPC и транспорт

Создание IPC-каналов

Обзор: 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 содержит службу, необходимую клиенту. Например, может быть неизвестно, на какой именно накопитель нужно будет записывать данные.

Статическое создание IPC-канала

Статический способ имеет следующие особенности:

- клиент и сервер находятся в остановленном состоянии в момент создания IPC-канала;
- создание инициируются родительским процессом, запускающим клиента и сервера (обычно это Einit);
- созданный IPC-канал невозможно удалить;
- чтобы получить IPC-дескриптор и идентификатор службы (riid) после создания IPC-канала, клиент и сервер должны использовать интерфейс локатора сервисов (`coresrv/sl/sl_api.h`).

Динамическое создание IPC-канала

Динамический способ имеет следующие особенности:

- клиент и сервер уже запущены в момент создания IPC-канала;
- создание инициируются совместно клиентом и сервером;
- созданный IPC-канал может быть удален;
- клиент и сервер получают IPC-дескриптор и идентификатор службы (riid) сразу после успешного создания IPC-канала.

Создание IPC-каналов с помощью init.yaml

Здесь собраны init-описания, демонстрирующие особенности создания IPC-каналов. Примеры задания свойств и аргументов процессов через init-описания разбираются в отдельной статье.

> В примерах в составе KasperskyOS Community Edition может использоваться формат init-описания с макросами (`init.yaml.in`).

Файл с init-описанием обычно называется `init.yaml`, хотя может иметь любое имя.

Соединение и запуск процесса-клиента и процесса-сервера

В следующем примере будут запущены два процесса: класса `Client` и 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. Процессы будут соединены IPC-каналом с именем `server_connection`.

init.yaml

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

```
    id: server_connection
- name: Server
```

Динамическое создание IPC-каналов

При динамическом создании IPC-канала используются функции:

- интерфейса сервера имен (Name Server);
- интерфейса менеджера соединений (Connection Manager).

Динамическое создание IPC-канала осуществляется по следующему сценарию:

1. Запускаются процессы: клиент, сервер и сервер имен.
2. Сервер подключается к серверу имен с помощью вызова `NsCreate()` и публикует имя сервера, имя интерфейса и имя службы с помощью вызова `NsPublishService()`.
3. Клиент подключается к серверу имен с помощью вызова `NsCreate()` и выполняет поиск имени сервера и имени службы по имени интерфейса с помощью вызова `NsEnumServices()`.
4. Клиент запрашивает доступ к службе с помощью вызова `KnCmConnect()`, передавая в качестве аргументов найденные имя сервера и имя службы.
5. Сервер вызывает функцию `KnCmListen()` для проверки наличия запросов на доступ к службе.
6. Сервер принимает запрос клиента на доступ к службе с помощью вызова `KnCmAccept()`, передавая в качестве аргументов имя клиента и имя службы, которые получены при вызове `KnCmListen()`.

> Пункты 2 и 3 могут быть опущены, если клиент заранее знает имя сервера и имя службы.

Сервер может снимать с публикации на сервере имен ранее опубликованные службы с помощью вызова `NsUnPublishService()`.

Сервер может отклонять запросы доступа к службам с помощью вызова `KnCmDrop()`.

> Для использования сервера имен политика безопасности решения должна разрешать взаимодействие процесса класса `kl.core.NameServer` и процессами, между которыми необходимо динамически создавать IPC-каналы.

Использование служб из состава KasperskyOS Community Edition

Добавление службы в решение

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 описанием интерфейсов процесса-получателя сообщения (сервера).

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

Фиксированная часть сообщения

Фиксированная часть сообщения содержит аргументы фиксированного размера, а также RIID и MID.

Аргументы фиксированного размера – это аргументы любых IDL-типов, кроме типа `sequence`.

RIID и MID идентифицируют вызываемый интерфейс и метод:

- RIID (Runtime Implementation ID) является порядковым номером вызываемой службы процесса (начиная с нуля).
- MID (Method ID) является порядковым номером метода в содержащем его интерфейсе (начиная с нуля).

Тип фиксированной части 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

Арена представляет собой буфер для хранения аргументов переменного размера (IDL-тип `sequence`).

Проверка структуры сообщения модулем безопасности

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

Формирование структуры сообщения

В составе KasperskyOS Community Edition поставляются следующие инструменты, позволяющие упростить для разработчика создание и упаковку IPC-сообщения:

- Библиотека `transport-kos` для работы с транспортом NkKosTransport.
- Компилятор NK, позволяющий сгенерировать специальные методы и типы.

Формирование простейшего IPC-сообщения показано в примерах echo и ping (`/opt/KasperskyOS-Community-Edition-<version>/examples/`).

Нахождение IPC-дескриптора

Клиентский и серверный IPC-дескрипторы требуется находить, 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 находить IPC-дескриптор. Вся работа с транспортом, включая нахождение IPC-дескрипторов, выполняется поставляемыми транспортными библиотеками.
> См. примеры gpio_*, net_*, net2_* и multi_vfs_* (`/opt/KasperskyOS-Community-Edition-<version>/examples/`).

Нахождение IPC-дескриптора при статическом создании канала

При статическом создании IPC-канала как клиент, так и сервер могут сразу после своего запуска узнать 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>
…
nk_iid_t iid;
Handle handle = ServiceLocatorRegister("server_connection", NULL, 0, &iid);
```

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

Нахождение 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`.

Нахождение идентификатора службы (riid)

Идентификатор службы (riid) требуется находить 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 аргумента 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 находить идентификатор службы. Вся работа с транспортом выполняется поставляемыми транспортными библиотеками.
> См. примеры gpio_*, net_*, net2_* и multi_vfs_* (`/opt/KasperskyOS-Community-Edition-<version>/examples/`).

Нахождение идентификатора службы при статическом создании канала

При статическом создании IPC-канала клиент может узнать идентификатор нужной службы с помощью метода `ServiceLocatorGetRiid()`, указав дескриптор IPC-канала и полное квалифицированное имя службы. Например, если экземпляр компонента `OpsComp` содержит 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 так же заголовочный файл `/opt/KasperskyOS-Community-Edition-<version>/sysroot-aarch64-kos/include/coresrv/sl/sl_api.h`.

Нахождение 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)
```

Библиотека libkos

Общие сведения о библиотеке libkos

Ядро KasperskyOS имеет ряд служб для управления дескрипторами, потоками, памятью, процессами, IPC-каналами, ресурсами ввода-вывода и т.д. Для доступа к службам используется библиотека `libkos`.

Библиотека libkos

Библиотека `libkos` состоит из двух частей:

- Первая часть предоставляет собой C-интерфейс для доступа к службам ядра KasperskyOS. Она доступна через заголовочные файлы, находящиеся в директории `coresrv`.
- Вторая часть библиотеки `libkos` предоставляет абстракции примитивов синхронизации, объектов и очередей. Она также содержит функции-обертки для более простой аллокации памяти и работы с потоками. Заголовочные файлы второй части `libkos` находятся в директории `kos`.

Библиотека `libkos` значительно упрощает использование служб ядра. Функции библиотеки `libkos` обеспечивают корректную упаковку IPC-сообщения и выполнение системных вызовов. Взаимодействие других библиотек (включая `libc`) с ядром происходит через библиотеку `libkos`.

Для использования службы ядра KasperskyOS нужно подключить соответствующий этой службе заголовочный файл библиотеки `libkos`. Например, для доступа к методам менеджера ввода-вывода (IO Manager) нужно подключить файл `io_api.h`:

```
#include <coresrv/io/io_api.h>
```

Файлы, используемые библиотекой libkos

Внутренняя реализация библиотеки `libkos` может использовать следующие файлы, экспортируемые ядром:

- Файлы на языке IDL (idl-описания). Содержат описания интерфейсов служб. Используются IPC-транспортом для корректной упаковки сообщений.
- Заголовочные файлы ядра. Эти файлы включаются библиотекой `libkos`.

Пример

Менеджер ввода-вывода (IO Manager) представлен для пользователя следующими файлами:

- `coresrv/io/io_api.h` – заголовочный файл библиотеки `libkos`;

- `services/io/IO.idl` – idl-описание менеджера ввода-вывода;
- `io/io_dma.h`, `io/io_irq.h` – заголовочные файлы ядра.

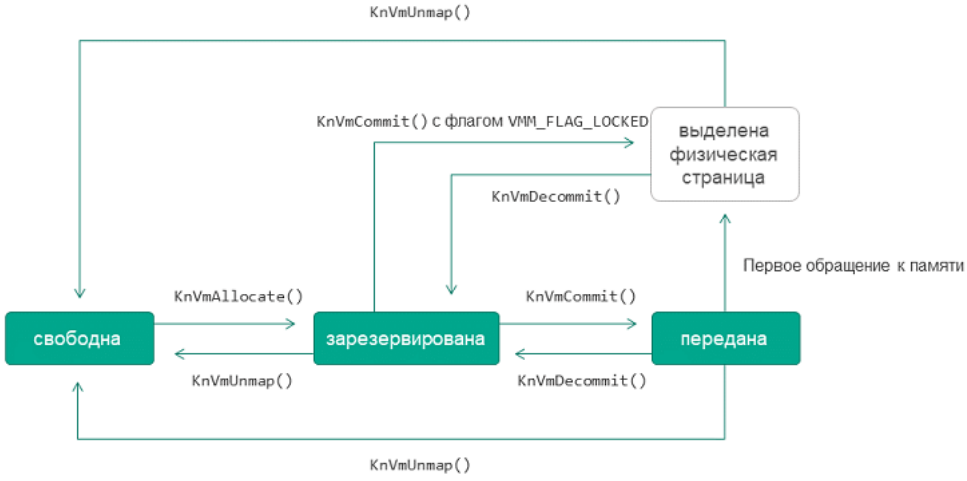
Память

Состояния памяти

Каждая страница виртуальной памяти может быть *свободна* (free), *зарезервирована* (reserved) или *передана* (committed).

Переход из свободного состояния в зарезервированное называется резервированием (аллокацией). Предварительное резервирование памяти (без передачи физических страниц) позволяет приложению заранее разметить свое адресное пространство. Обратный переход из зарезервированного в свободное состояние называется освобождением памяти.

Назначение физической памяти для ранее зарезервированной страницы виртуальной памяти называется передачей памяти, а обратный переход из переданного состояния в зарезервированное – возвращением памяти.



Переходы между состояниями страницы памяти

KnVmAllocate()

Функция объявлена в файле `coresrv/vmm/vmm_api.h`.

```
void *KnVmAllocate(void *addr, rtl_size_t size, int flags);
```

Функция резервирует диапазон физических страниц, задаваемый параметрами `addr` и `size`. Если указан флаг **VMM_FLAG_COMMIT**, функция резервирует и передает страницы за один вызов.

Параметры:

- `addr` – странично-выровненный базовый физический адрес; если задать `addr` равным **0**, система сама выберет свободный участок физической памяти;
- `size` – размер участка памяти в байтах (должен быть кратен размеру страницы);
- `flags` – флаги аллокации.

Функция возвращает базовый виртуальный адрес зарезервированного участка. Если зарезервировать участок памяти невозможно, функция возвращает **RTL_NULL**.

Флаги аллокации

В параметре `flags` можно использовать следующие флаги (`vmm/flags.h`):

- **VMM_FLAG_RESERVE** – обязательный флаг;
- **VMM_FLAG_COMMIT** – позволяет за один вызов `KnVmAllocate()` резервировать и передать страницы памяти в "ленивом" режиме;
- **VMM_FLAG_LOCKED** – используется совместно с **VMM_FLAG_COMMIT**; позволяет сразу передать физические страницы памяти вместо "ленивой" передачи;
- **VMM_FLAG_WRITE_BACK**, **VMM_FLAG_WRITE_THROUGH**, **VMM_FLAG_WRITE_COMBINE**, **VMM_FLAG_CACHE_DISABLE** и **VMM_FLAG_CACHE_MASK** – управляют кэшированием страниц памяти;
- **VMM_FLAG_READ**, **VMM_FLAG_WRITE**, **VMM_FLAG_EXECUTE** и **VMM_FLAG_RWX_MASK** – атрибуты защиты памяти;
- **VMM_FLAG_LOW_GUARD** и **VMM_FLAG_HIGH_GUARD** – добавление защитной страницы перед и после выделенной памяти соответственно;
- **VMM_FLAG_GROW_DOWN** – определение направления доступа к памяти (от старших адресов к младшим).

Допустимые комбинации атрибутов защиты памяти:

- **VMM_FLAG_READ** – разрешено чтение содержимого страницы;
- **VMM_FLAG_READ | VMM_FLAG_WRITE** – разрешено чтение и изменение содержимого страницы;
- **VMM_FLAG_READ | VMM_FLAG_EXECUTE** – разрешено чтение и выполнение содержимого страницы;
- **VMM_FLAG_RWX_MASK** или **VMM_FLAG_READ | VMM_FLAG_WRITE | VMM_FLAG_EXECUTE** – полный доступ к содержимому страницы (эти записи эквивалентны).

Пример

```
coredump->base = KnVmAllocate(RTL_NULL, vmaSize,
                              VMM_FLAG_READ | VMM_FLAG_RESERVE |
                              VMM_FLAG_WRITE | VMM_FLAG_COMMIT |
                              VMM_FLAG_LOCKED);
```

При необходимости можно изменить заданные атрибуты защиты участка памяти с помощью функции `KnVmProtect()`.

KnVmCommit()

Функция объявлена в файле `coresrv/vmm/vmm_api.h`.

```
Retcode KnVmCommit(void *addr, rtl_size_t size, int flags);
```

Функция передает диапазон физических страниц, задаваемый параметрами addr и size.

Все передаваемые страницы должны быть предварительно зарезервированы.

Параметры:

- `addr` – странично-выровненный базовый виртуальный адрес участка памяти;
- `size` – размер участка памяти в байтах (должен быть кратен размеру страницы);
- `flags` – параметр не используется (укажите флаг **VMM_FLAG_LOCKED** в значении параметра для обеспечения совместимости).

В случае успешной передачи страниц функция возвращает **rcOk**.

KnVmDecommit()

Функция объявлена в файле `coresrv/vmm/vmm_api.h`.

```
Retcode KnVmDecommit(void *addr, rtl_size_t size);
```

Функция освобождает диапазон страниц (переводит их в зарезервированное состояние).

Параметры:

- `addr` – странично-выровненный базовый виртуальный адрес участка памяти;
- `size` – размер участка памяти в байтах (должен быть кратен размеру страницы).

В случае успешного освобождения страниц функция возвращает **rcOk**.

KnVmProtect()

Функция объявлена в файле `coresrv/vmm/vmm_api.h`.

```
Retcode KnVmProtect(void *addr, rtl_size_t size, int newFlags);
```

Функция изменяет атрибуты защиты зарезервированных или переданных страниц памяти.

Параметры:

- `addr` – странично-выровненный базовый виртуальный адрес участка памяти;
- `size` – размер участка памяти в байтах (должен быть кратен размеру страницы);
- `newFlags` – новые атрибуты защиты.

В случае успешного изменения атрибутов защиты функция возвращает **rcOk**.

Допустимые комбинации атрибутов защиты памяти:

- **VMM_FLAG_READ** – разрешено чтение содержимого страницы;
- **VMM_FLAG_READ | VMM_FLAG_WRITE** – разрешено чтение и изменение содержимого страницы;
- **VMM_FLAG_READ | VMM_FLAG_EXECUTE** – разрешено чтение и выполнение содержимого страницы;
- **VMM_FLAG_RWX_MASK** или **VMM_FLAG_READ | VMM_FLAG_WRITE | VMM_FLAG_EXECUTE** – полный доступ к содержимому страницы (эти записи эквивалентны).

KnVmUnmap()

Функция объявлена в файле `coresrv/vmm/vmm_api.h`.

```
Retcode KnVmUnmap(void *addr, rtl_size_t size);
```

Функция освобождает участок памяти.

Параметры:

- `addr` – странично-выровненный адрес участка памяти;
- `size` – размер участка памяти.

В случае успешного освобождения страниц функция возвращает **rcOk**.

Аллокация памяти

KosMemAlloc()

Функция объявлена в файле `kos/alloc.h`.

```
void *KosMemAlloc(rtl_size_t size);
```

Функция выделяет (резервирует и передает) участок памяти размером `size` байт.

Функция возвращает указатель на выделенный участок или **RTL_NULL**, если память не удалось выделить.

> Память, выделенная с помощью функции `KosMemAlloc()`, имеет следующие флаги аллокации: **VMM_FLAG_READ | VMM_FLAG_WRITE**, **VMM_FLAG_RESERVE**, **VMM_FLAG_COMMIT**, **VMM_FLAG_LOCKED**. Чтобы выделить память с другими флагами аллокации, используйте функцию `KnVmAllocate()`.

KosMemAllocEx()

Функция объявлена в файле `kos/alloc.h`.

```
void *KosMemAllocEx(rtl_size_t size, rtl_size_t align, int zeroed);
```

Функция аналогична `KosMemAlloc()`, но при этом имеет дополнительные параметры:

- `align` – выравнивание участка памяти в байтах (степень двойки);
- `zeroed` – нужно ли заполнить участок памяти нулями (**1** – заполнить, **0** – не заполнять).

KosMemFree()

Функция объявлена в файле `kos/alloc.h`.

```
void KosMemFree(void *ptr);
```

Функция освобождает участок памяти, выделенный с помощью функции `KosMemAlloc()`, `KosMemZalloc()` или `KosMemAllocEx()`.

- `ptr` – указатель на освобождаемый участок памяти.

KosMemGetSize()

Функция объявлена в файле `kos/alloc.h`.

```
rtl_size_t KosMemGetSize(void *ptr);
```

Функция возвращает размер (в байтах) участка памяти, выделенного с помощью функции `KosMemAlloc()`, `KosMemZalloc()` или `KosMemAllocEx()`.

- `ptr` – указатель на участок памяти.

KosMemZalloc()

Функция объявлена в файле `kos/alloc.h`.

```
void *KosMemZalloc(rtl_size_t size);
```

Функция аналогична `KosMemAlloc()`, но при этом заполняет выделяемый участок памяти нулями.

Потоки

KosThreadCallback()

Прототип callback-функции объявлен в файле `kos/thread.h`.

```
typedef void KosThreadCallback(KosThreadCallbackReason reason);

/* Аргумент callback-функции */
typedef enum KosThreadCallbackReason {
    KosThreadCallbackReasonCreate,
    KosThreadCallbackReasonDestroy,
} KosThreadCallbackReason;
```

При создании нового потока все зарегистрированные callback-функции будут вызваны с аргументом `KosThreadCallbackReasonCreate`, при завершении – с аргументом `KosThreadCallbackReasonDestroy`.

KosThreadCallbackRegister()

Функция объявлена в файле `kos/thread.h`.

```
Retcode KosThreadCallbackRegister(KosThreadCallback *callback);
```

Функция регистрирует пользовательскую callback-функцию. При создании и завершении потока будут вызваны все зарегистрированные callback-функции.

KosThreadCallbackUnregister()

Функция объявлена в файле `kos/thread.h`.

```
Retcode KosThreadCallbackUnregister(KosThreadCallback *callback);
```

Функция дерегистрирует (удаляет из списка вызываемых) пользовательскую callback-функцию.

KosThreadCreate()

Функция объявлена в файле `kos/thread.h`.

```
Retcode KosThreadCreate(Tid                *tid,
                        rtl_uint32_t        priority,
                        rtl_uint32_t        stackSize,
                        ThreadRoutine       routine,
                        void               *context,
                        int                 suspended);
```

Функция создает новый поток.

Входные параметры:

- `priority` – приоритет, должен быть в интервале от **0** до **31**; доступны следующие константы приоритета: `ThreadPriorityLowest` (**0**), `ThreadPriorityNormal` (**15**) и `ThreadPriorityHighest` (**31**);
- `stackSize` – размер стека;
- `routine` – функция, которая будет выполняться в потоке;
- `context` – аргумент, который будет передан в функцию `routine`;
- `suspended` – позволяет создать поток в приостановленном состоянии (**1** – создать приостановленный, **0** – не приостановленный).

Выходные параметры:

- `tid` – идентификатор созданного потока.

Пример

```
int main(int argc, char **argv)
{
    Tid tidB;
    Tid tidC;
    Retcode rcB;
    Retcode rcC;

    static ThreadContext threadContext[] = {
        {.ddi = "B", .deviceName = "/pci/bus0/dev2/fun0/DDI_B"},
        {.ddi = "C", .deviceName = "/pci/bus0/dev2/fun0/DDI_C"},
    };

    rcB = KosThreadCreate(&tidB, ThreadPriorityNormal,
                          ThreadStackSizeDefault,
                          FbHotplugThread,
                          &threadContext[0], 0);
    if (rcB != rcOk)
        ERR("Failed to start thread %s", threadContext[0].ddi);
```

```
    rcC = KosThreadCreate(&tidC, ThreadPriorityNormal,
                          ThreadStackSizeDefault,
                          FbHotplugThread,
                          &threadContext[1], 0);

    if (rcC != rcOk)
        ERR("Failed to start thread %s", threadContext[1].ddi);

    /* Ожидание завершения потоков */
    ...
}
```

KosThreadCurrentId()

Функция объявлена в файле `kos/thread.h`.

```
Tid KosThreadCurrentId(void);
```

Функция запрашивает TID вызывающего потока.

В случае успеха функция возвращает идентификатор потока (TID).

KosThreadExit()

Функция объявлена в файле `kos/thread.h`.

```
void KosThreadExit(rtl_int32_t exitCode);
```

Функция принудительно завершает текущий поток с кодом выхода `exitCode`.

KosThreadGetStack()

Функция объявлена в файле `kos/thread.h`.

```
void *KosThreadGetStack(Tid tid, rtl_uint32_t *size);
```

Функция получает стек потока с идентификатором `tid`.

Выходной параметр `size` содержит размер стека.

В случае успеха функция возвращает указатель на начало стека.

KosThreadOnce()

Функция объявлена в файле `kos/thread.h`.

```
typedef int KosThreadOnceState;

Retcode KosThreadOnce(KosThreadOnceState       *onceControl,
                      void                   (* initRoutine) (void));
```

Функция позволяет вызвать заданную процедуру `initRoutine` в точности один раз, даже при вызове из нескольких потоков.

Параметр `onceControl` предназначен для контроля однократного вызова процедуры.

При успешном вызове процедуры, а также если она уже была вызвана ранее, функция `KosThreadOnce()` возвращает **rcOk**.

KosThreadResume()

Функция объявлена в файле `kos/thread.h`.

```
Retcode KosThreadResume(Tid tid);
```

Функция возобновляет поток с идентификатором `tid`, созданный в приостановленном состоянии.

В случае успеха функция возвращает **rcOk**.

KosThreadSleep()

Функция объявлена в файле `kos/thread.h`.

```
Retcode KosThreadSleep(rtl_uint32_t mdelay);
```

Функция приостанавливает выполнение текущего потока на `mdelay` миллисекунд.

В случае успеха функция возвращает **rcOk**.

KosThreadSuspend()

Функция объявлена в файле `kos/thread.h`.

```
Retcode KosThreadSuspend(Tid tid);
```

Функция необратимо останавливает текущий поток, не завершая его.

Параметр `tid` должен быть равен идентификатору текущего потока (ограничение текущей имплементации).

В случае успеха функция возвращает **rcOk**.

KosThreadTerminate()

Функция объявлена в файле `kos/thread.h`.

```
Retcode KosThreadTerminate(Tid tid, rtl_int32_t exitCode);
```

Функция завершает поток вызывающего процесса. Параметр `tid` задает идентификатор потока.

Если `tid` указывает на текущий поток, то параметр `exitCode` задает код выхода потока.

В случае успеха функция возвращает **rcOk**.

KosThreadTlsGet()

Функция объявлена в файле `kos/thread.h`.

```
void *KosThreadTlsGet(void);
```

Функция возвращает указатель на локальное хранилище потока (TLS) или **RTL_NULL**, если TLS отсутствует.

KosThreadTlsSet()

Функция объявлена в файле `kos/thread.h`.

```
Retcode KosThreadTlsSet(void *tls);
```

Функция задает адрес локального хранилища потока (TLS).

Входной аргумент `tls` содержит адрес TLS.

KosThreadWait()

Функция объявлена в файле `kos/thread.h`.

```
int KosThreadWait(rtl_uint32_t tid, rtl_uint32_t timeout);
```

Функция приостанавливает выполнение текущего потока до момента завершения потока с идентификатором `tid` или до истечения `timeout` миллисекунд.

Вызов `KosThreadWait()` с нулевым значением `timeout` аналогичен вызову `KosThreadYield()`.

В случае успеха функция возвращает **rcOk**, в случае таймаута **rcTimeout**.

KosThreadYield()

Функция объявлена в файле `kos/thread.h`.

```
void KosThreadYield(void);
```

Функция передает выполнение вызвавшего ее потока следующему потоку.

Вызов `KosThreadYield()` аналогичен вызову `KosThreadSleep()` с нулевым значением `mdelay`.

Дескрипторы

KnHandleClose()

Функция объявлена в файле `coresrv/handle/handle_api.h`.

```
Retcode KnHandleClose(Handle handle);
```

Функция удаляет дескриптор `handle`.

В случае успеха функция возвращает **rcOk**, иначе возвращает код ошибки.

> Удаление дескриптора не делает недействительными его предков и потомков (в отличие от отзыва дескриптора, который делает недействительными его потомков – см. `KnHandleRevoke()` и `KnHandleRevokeSubtree()`). Удаление дескриптора также не нарушает целостность дерева наследования дескрипторов. Место удаленного дескриптора занимает его предок – он становится непосредственным предком потомков удаленного дескриптора.

KnHandleCreateBadge()

Функция объявлена в файле `coresrv/handle/handle_api.h`.

```
Retcode KnHandleCreateBadge(Notice notice, rtl_uintptr_t eventId,
                            void *context, Handle *handle);
```

Функция создает объект контекста передачи ресурса для контекста передачи ресурса `context` и настраивает приемник уведомлений `notice` на прием уведомлений об этом объекте. Приемник уведомлений настраивается на прием уведомлений о событиях, которые соответствуют флагам маски событий `EVENT_OBJECT_DESTROYED` и `EVENT_BADGE_CLOSED`.

Входной параметр `eventId` задает идентификатор записи вида "ресурс – маска событий" в приемнике уведомлений.

Выходной параметр `handle` содержит дескриптор объекта контекста передачи ресурса.

В случае успеха функция возвращает **rcOk**, иначе возвращает код ошибки.

KnHandleCreateUserObject()

Функция объявлена в файле `coresrv/handle/handle_api.h`.

```
Retcode KnHandleCreateUserObject(rtl_uint32_t type, rtl_uint32_t rights,
                                 void *context, Handle *handle);
```

Функция создает дескриптор `handle` типа `type` с маской прав `rights`.

Параметр `type` может принимать значения от `HANDLE_TYPE_USER_FIRST` до `HANDLE_TYPE_USER_LAST`.

Макросы `HANDLE_TYPE_USER_FIRST` и `HANDLE_TYPE_USER_LAST` определены в заголовочном файле `handle/handletype.h`.

Параметр `context` задает контекст пользовательского ресурса. В случае успеха функция возвращает **rcOk**, иначе возвращает код ошибки.

Пример

```
Retcode ServerPortInit(ServerPort *serverPort)
{
    Retcode         rc          = rcInvalidArgument;
    Notice   serverEventNotice;

    rc = KnHandleCreateUserObject(HANDLE_TYPE_USER_FIRST, OCAP_HANDLE_SET_EVENT |
OCAP_HANDLE_GET_EVENT,
                                  serverPort, &serverPort->handle);
    if (rc == rcOk) {
        KosRefObject(serverPort);
        rc = KnNoticeSubscribeToObject(serverEventNotice,
                                       serverPort->handle,
                                       EVENT_OBJECT_DESTROYED,
                                       (rtl_uintptr_t) serverPort);
        if (rc != rcOk) {
            KosPutObject(serverPort);
            KnHandleClose(serverPort->handle);
            serverPort->handle = INVALID_HANDLE;
        }
    }

    return rc;
}
```

KnHandleRevoke()

Функция объявлена в файле `coresrv/handle/handle_api.h`.

```
Retcode KnHandleRevoke(Handle handle);
```

Функция удаляет дескриптор `handle` и отзывает всех его потомков.

В случае успеха функция возвращает **rcOk**, иначе возвращает код ошибки.

> Отзыв дескрипторов не удаляет их, но через отозванные дескрипторы невозможно обращаться к ресурсам. Любая функция, которая принимает дескриптор, завершается с ошибкой **rcHandleRevoked**, если эта функция вызывается с отозванным дескриптором.

KnHandleRevokeSubtree()

Функция объявлена в файле `coresrv/handle/handle_api.h`.

```
Retcode KnHandleRevokeSubtree(Handle handle, Handle badge);
```

Функция отзывает дескрипторы, которые образуют поддерево наследования дескриптора `handle`.

Корневым узлом поддерева наследования является дескриптор, который порожден передачей дескриптора `handle` в ассоциации с объектом контекста передачи ресурса `badge`.

В случае успеха функция возвращает **rcOk**, иначе возвращает код ошибки.

> Отзыв дескрипторов не удаляет их, но через отозванные дескрипторы невозможно обращаться к ресурсам. Любая функция, которая принимает дескриптор, завершается с ошибкой **rcHandleRevoked**, если эта функция вызывается с отозванным дескриптором.

nk_get_badge_op()

Функция объявлена в файле `nk/types.h`.

```
static inline
nk_err_t nk_get_badge_op(const nk_handle_desc_t *desc,
                         nk_rights_t operation,
                         nk_badge_t *badge);
```

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.

nk_is_handle_dereferenced()

Функция объявлена в файле `nk/types.h`.

```
static inline
nk_bool_t nk_is_handle_dereferenced(const nk_handle_desc_t *desc);
```

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.

Управление дескрипторами

Для управления дескрипторами используются функции менеджера дескрипторов (Handle Manager) и подсистемы уведомлений (Notification Subsystem).

Менеджер дескрипторов представлен для пользователя следующими файлами:

- `coresrv/handle/handle_api.h` – заголовочный файл библиотеки `libkos`;
- `services/handle/Handle.idl` – описание IPC-интерфейса менеджера дескрипторов на языке IDL.

Подсистема уведомлений представлена для пользователя следующими файлами:

- `coresrv/handle/notice_api.h` – заголовочный файл библиотеки `libkos`;
- `services/handle/Notice.idl` – описание IPC-интерфейса подсистемы уведомлений на языке IDL.

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 `services/ocap.h`). 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 `services/ocap.h`. Структура специальной части для пользовательских ресурсов определяется поставщиком ресурсов с использованием макроса `OCAP_HANDLE_SPEC()`, который определен в заголовочном файле `services/ocap.h`. Поставщику ресурсов необходимо экспортировать публичные заголовочные файлы с описанием структуры специальной части.

При создании дескриптора системного ресурса маска прав задается ядром KasperskyOS, которое применяет маски прав из заголовочного файла `services/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

Дескрипторы пользовательских ресурсов создаются поставщиками ресурсов. Для создания дескрипторов пользовательских ресурсов используется функция `KnHandleCreateUserObject()`, которая объявлена в заголовочном файле `coresrv/handle/handle_api.h`.

handle_api.h (фрагмент)

```
/**
 * Функция создает дескриптор handle типа type с маской прав rights.
 * Параметр type может принимать значения от HANDLE_TYPE_USER_FIRST до
 * HANDLE_TYPE_USER_LAST. Макросы HANDLE_TYPE_USER_FIRST и HANDLE_TYPE_USER_LAST
 * определены в заголовочном файле handletype.h. Параметр context задает контекст
 * пользовательского ресурса.
 * В случае успеха функция возвращает rcOk, иначе возвращает код ошибки.
 */
Retcode KnHandleCreateUserObject(rtl_uint32_t type, rtl_uint32_t rights,
                                 void *context, Handle *handle);
```

Контекст пользовательского ресурса – данные, позволяющие поставщику ресурса идентифицировать ресурс и его состояние, когда запрашивается доступ к ресурсу другими программами. 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.

> Параметр `type` функции `KnHandleCreateUserObject()` зарезервирован для возможного использования в будущем и не влияет на ее поведение, но должен принимать значение из интервала, указанного в комментарии к функции.

О маске прав дескриптора см. "Маска прав дескриптора".

Передача дескрипторов

Общие сведения

Передача дескрипторов между программами осуществляется для того, чтобы *клиенты* (программы, которые используют ресурсы) получали доступ к требуемым ресурсам. По причине локальности дескрипторов передача дескриптора инициирует на стороне принимающей программы создание дескриптора из ее пространства дескрипторов. Этот дескриптор регистрируется как потомок отправленного дескриптора и идентифицирует тот же ресурс.

Один дескриптор может быть передан многократно одной или нескольким программам. Каждая передача порождает нового потомка переданного дескриптора на стороне принимающей программы. Программа может передавать дескрипторы, которые она получила от других программ или ядра KasperskyOS (при создании дескрипторов системных ресурсов). Поэтому у дескриптора может быть несколько поколений потомков. Иерархия порождения дескрипторов для каждого ресурса хранится в ядре KasperskyOS в виде *дерева наследования дескрипторов*.

Программа может передавать дескрипторы как пользовательских, так и системных ресурсов, если права доступа этих дескрипторов разрешают выполнять передачу. У потомка может быть меньше прав доступа, чем у предка. Например, передающая программа имеет права доступа к файлу на чтение и запись, а передает права доступа только на чтение. Передающая программа также может запретить принимающей программе дальнейшую передачу дескриптора. Права доступа задаются в передаваемой маске прав дескриптора.

Условия для передачи дескрипторов

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

1. Между программами создан IPC-канал.
2. Политика безопасности решения (`security.psl`) разрешает взаимодействие программ.
3. Реализованы интерфейсные методы для передачи дескрипторов.
4. Программа-клиент получила идентификатор службы (RIID) программы-сервера с методами для передачи дескрипторов.

Интерфейсные методы для передачи дескрипторов объявляются в IDL-описании с входными (`in`) и/или выходными (`out`) параметрами типа `Handle`. Методы с входными параметрами типа `Handle` предназначены для передачи дескрипторов от программы-клиента программе-серверу. Методы с выходными параметрами типа `Handle` предназначены для передачи дескрипторов от программы-сервера программе-клиенту. Для одного метода может быть объявлено не более семи входных и семи выходных параметров типа `Handle`.

Пример IDL-описания, где объявлены интерфейсные методы 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 запросов `*_req` и/или ответов `*_res` поле типа `nk_handle_desc_t` (далее также *транспортный контейнер дескриптора*). Этот тип объявлен в заголовочном файле `nk/types.h` 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 программе-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. Программа-сервер получает указатель на контекст передачи ресурса при разыменовании дескриптора.

Программа-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 запрос.

Если программа-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.

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

Чтобы ассоциировать передачу дескриптора с контекстом передачи ресурса, программа-сервер помещает в поле `badge` структуры `nk_handle_desc_t` дескриптор объекта контекста передачи ресурса. *Объект контекста передачи ресурса* – объект, в котором хранится указатель на контекст передачи ресурса. Объект контекста передачи ресурса создается функцией `KnHandleCreateBadge()`, которая объявлена в заголовочном файле `coresrv/handle/handle_api.h`. Работа этой функции связана с подсистемой уведомлений о состоянии ресурсов, так как программе-серверу нужно знать, когда объект контекста передачи ресурса будет закрыт и прекратит свое существование. Эти сведения требуются программе-серверу, чтобы освободить или использовать повторно память, которая отведена для хранения контекста передачи ресурса.

Объект контекста передачи ресурса будет закрыт, когда будут удалены или отозваны потомки дескриптора (см. "Удаление дескрипторов", "Отзыв дескрипторов"), которые порождены его передачей в ассоциации с этим объектом. (Переданный дескриптор может быть удален не только целенаправленно, но и, например, при неожиданном завершении работы принимающей программы-клиента.) Получив уведомление о закрытии объекта контекста передачи ресурса, программа-сервер удаляет дескриптор этого объекта. После этого объект контекста передачи ресурса прекращает свое существование. Получив уведомление, что объект контекста передачи ресурса прекратил свое существование, программа-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.

handle_api.h (фрагмент)

```
/**
 * Функция создает объект контекста передачи ресурса для
 * контекста передачи ресурса context и настраивает
 * приемник уведомлений notice на прием уведомлений об
 * этом объекте. Приемник уведомлений настраивается на
 * прием уведомлений о событиях, которые соответствуют
 * флагам маски событий OBJECT_DESTROYED и EVENT_BADGE_CLOSED.
```

```
 * Входной параметр eventId задает идентификатор записи
 * вида "ресурс – маска событий" в приемнике уведомлений.
 * Выходной параметр handle содержит дескриптор объекта
 * контекста передачи ресурса.
 * В случае успеха функция возвращает rcOk, иначе возвращает код ошибки.
 */
Retcode KnHandleCreateBadge(Notice notice, rtl_uintptr_t eventId,
                            void *context, Handle *handle);
```

Упаковка данных в транспортный контейнер дескриптора

Для упаковки дескриптора, маски прав дескриптора и дескриптора объекта контекста передачи ресурса в транспортный контейнер дескриптора используется макрос `nk_handle_desc()`, который определен в заголовочном файле `nk/types.h`. Этот макрос принимает переменное число аргументов.

Если не передавать макросу ни одного аргумента, то в поле дескриптора `handle` структуры `nk_handle_desc_t` будет записано значение `NK_INVALID_HANDLE`.

Если передать макросу один аргумент, то этот аргумент интерпретируется как дескриптор.

Если передать макросу два аргумента, то первый аргумент интерпретируется как дескриптор, второй аргумент интерпретируется как маска прав дескриптора.

Если передать макросу три аргумента, то первый аргумент интерпретируется как дескриптор, второй аргумент интерпретируется как маска прав дескриптора, третий аргумент интерпретируется как дескриптор объекта контекста передачи ресурса.

Извлечение данных из транспортного контейнера дескриптора

Для извлечения дескриптора, маски прав дескриптора и указателя на контекст передачи ресурса из транспортного контейнера дескриптора используются соответственно функции `nk_get_handle()`, `nk_get_rights()` и `nk_get_badge_op()` (или `nk_get_badge()`), которые определены в заголовочном файле `nk/types.h`. Функции `nk_get_badge_op()` и `nk_get_badge()` используются только при разыменовании дескрипторов.

Сценарии передачи дескрипторов

Сценарий передачи дескрипторов от программы-клиента программе-серверу включает следующие шаги:

1. Передающая программа-клиент упаковывает дескрипторы и маски прав дескрипторов в поля структуры запросов `*_req` типа `nk_handle_desc_t`.

2. Передающая программа-клиент вызывает интерфейсный метод для передачи дескрипторов программе-серверу. Этот метод выполняет системный вызов `Call()`.

3. Принимающая программа-сервер получает запрос, выполняя системный вызов `Recv()`.

4. Диспетчер на стороне принимающей программы-сервера вызывает метод, который соответствует запросу. Этот метод извлекает дескрипторы и маски прав дескрипторов из полей структуры запросов `*_req` типа `nk_handle_desc_t`.

Сценарий передачи дескрипторов от программы-сервера программе-клиенту включает следующие шаги:

1. Принимающая программа-клиент вызывает интерфейсный метод для получения дескрипторов от программы-сервера. Этот метод выполняет системный вызов `Call()`.

2. Передающая программа-сервер получает запрос, выполняя системный вызов `Recv()`.

3. Диспетчер на стороне передающей программы-сервера вызывает метод, который соответствует запросу. Этот метод упаковывает дескрипторы, маски прав дескрипторов и дескрипторы объектов контекстов передачи ресурсов в поля структуры ответов `*_res` типа `nk_handle_desc_t`.

4. Передающая программа-сервер отвечает на запрос, выполняя системный вызов `Reply()`.

5. На стороне принимающей программы-клиента интерфейсный метод возвращает управление. После этого принимающая программа-клиент извлекает дескрипторы и маски прав дескрипторов из полей структуры ответов `*_res` типа `nk_handle_desc_t`.

Если передающая программа задает в передаваемой маске прав дескриптора больше прав доступа, чем задано для передаваемого дескриптора (владельцем которого она является), то передача не осуществляется. В этом случае выполнение системного вызова `Call()` передающей или принимающей программой-клиентом, а также выполнение системного вызова `Reply()` передающей программой-сервером завершается с ошибкой `rcSecurityDisallow`.

Разыменование дескрипторов

Разыменование дескриптора – это операция, при которой программа-клиент отправляет программе-серверу дескриптор, а программа-сервер получает указатель на контекст передачи ресурса, маску прав отправленного дескриптора и предка отправленного программой-клиентом дескриптора, которым программа-сервер уже владеет. Разыменование выполняется, когда программа-клиент, вызывая методы работы с ресурсом (например, чтения, записи, закрытия доступа), передает программе-серверу дескриптор, который был получен от этой программы-сервера 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. Программа-клиент упаковывает дескриптор в поле структуры запросов `*_req` типа `nk_handle_desc_t`.

2. Программа-клиент вызывает интерфейсный метод для отправки дескриптора программе-серверу с целью выполнения действий с ресурсом. Этот метод выполняет системный вызов `Call()`.

3. Программа-сервер принимает запрос, выполняя системный вызов `Recv()`.

4. Диспетчер на стороне программы-сервера вызывает метод, который соответствует 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 программой-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 программа-сервер использует функции `nk_is_handle_dereferenced()` и `nk_get_badge_op()`, которые объявлены в заголовочном файле `nk/types.h`.

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

Отзыв дескрипторов

Программа может отзывать потомков дескриптора, которым она владеет. Отзыв дескрипторов осуществляется согласно дереву наследования дескрипторов.

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

Отзыв выполняется функциями `KnHandleRevoke()` и `KnHandleRevokeSubtree()`, которые объявлены в заголовочном файле `coresrv/handle/handle_api.h`. Функция `KnHandleRevokeSubtree()` использует объект контекста передачи ресурса, который создается при передаче дескрипторов.

handle_api.h (фрагмент)

```
/**
 * Функция удаляет дескриптор handle и отзывает всех его потомков.
 * В случае успеха функция возвращает rcOk, иначе возвращает код ошибки.
 */
Retcode KnHandleRevoke(Handle handle);

/**
 * Функция отзывает дескрипторы, которые образуют поддерево
 * наследования дескриптора handle. Корневым узлом поддерева
 * наследования является дескриптор, который порожден передачей
 * дескриптора handle в ассоциации с объектом контекста
 * передачи ресурса badge.
 * В случае успеха функция возвращает rcOk, иначе возвращает код ошибки.
 */
Retcode KnHandleRevokeSubtree(Handle handle, Handle badge);
```

Уведомление о состоянии ресурсов

Программы могут отслеживать события, которые происходят с ресурсами (как системными, так и пользовательскими), а также информировать о событиях с пользовательскими ресурсами другие программы.

Функции подсистемы уведомлений объявлены в заголовочном файле `coresrv/handle/notice_api.h`. Подсистема уведомлений предусматривает использование масок 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 (флаги этих событий определены в заголовочном файле `handle/event_descr.h`). Например, в общей части находится флаг `EVENT_OBJECT_DESTROYED`, который определяет событие "прекращение существования ресурса". Специальная часть описывает события, специфичные для пользовательского ресурса. Структура специальной части определяется поставщиком ресурса с использованием макроса `OBJECT_EVENT_SPEC()`, который определен в заголовочном файле `handle/event_descr.h`. Поставщику ресурса необходимо экспортировать публичные заголовочные файлы с описанием структуры специальной части.

Сценарий получения уведомлений о событиях, которые происходят с ресурсом, включает следующие шаги:

1. Функцией `KnNoticeCreate()` создается *приемник уведомлений* (объект, в котором накапливаются уведомления).

2. В приемник уведомлений функцией `KnNoticeSubscribeToObject()` добавляются записи вида "ресурс – маска событий", чтобы настроить его на прием уведомлений о событиях, которые происходят с интересующими ресурсами. Набор отслеживаемых событий задается для каждого ресурса маской событий.

3. Чтобы извлекать уведомления из приемника уведомлений, вызывается функция `KnNoticeGetEvent()`.

Чтобы уведомлять программы о событиях, которые происходят с пользовательским ресурсом, используется функция `KnNoticeSetObjectEvent()`. Вызов этой функции инициирует появление соответствующих уведомлений в приемниках уведомлений, которые настроены на отслеживание этих событий с этим ресурсом.

notice_api.h (фрагмент)

```
/**
 * Функция создает приемник уведомлений notice.
 * В случае успеха функция возвращает rcOk, иначе возвращает код ошибки.
 */
Retcode KnNoticeCreate(Notice *notice);

/**
 * Функция добавляет запись вида "ресурс – маска событий"
 * в приемник уведомлений notice, чтобы он принимал уведомления о
 * событиях, которые происходят с ресурсом object и соответствуют
 * маске событий evMask. Входной параметр evId задает идентификатор
 * записи, который назначается пользователем и используется, чтобы
 * идентифицировать запись в полученных уведомлениях.
 * В случае успеха функция возвращает rcOk, иначе возвращает код ошибки.
 */
Retcode KnNoticeSubscribeToObject(Notice notice,
                                  Handle object,
                                  rtl_uint32_t evMask,
                                  rtl_uintptr_t evId);
```

```
/**
 * Функция извлекает уведомления из приемника уведомлений notice,
 * ожидая наступления событий в течение msec миллисекунд.
 * Входной параметр countMax задает максимальное число
 * уведомлений, которое может быть извлечено. Выходной параметр
 * events содержит набор извлеченных уведомлений типа EventDesc.
 * Выходной параметр count содержит число уведомлений, которые
 * были извлечены.
 * В случае успеха функция возвращает rcOk, иначе возвращает код ошибки.
 */
Retcode KnNoticeGetEvent(Notice notice,
                         rtl_uint64_t msec,
                         rtl_size_t countMax,
                         EventDesc *events,
                         rtl_size_t *count);

/* Структура уведомления */
typedef struct {
    /* Идентификатор записи "ресурс – маска событий"
     * в приемнике уведомлений */
    rtl_uintptr_t   eventId;
    /* Маска событий, которые произошли. */
    rtl_uint32_t    eventMask;
} EventDesc;

/**
 * Функция сигнализирует, что события из маски событий
 * evMask произошли с пользовательским ресурсом object.
 * Нельзя устанавливать флаги общей части маски событий,
 * так как о событиях из общей части маски событий может
 * сигнализировать только ядро KasperskyOS.
 * В случае успеха функция возвращает rcOk, иначе возвращает код ошибки.
 */
Retcode KnNoticeSetObjectEvent(Handle object, rtl_uint32_t evMask);
```

Удаление дескрипторов

Программа может удалять дескрипторы, которыми она владеет. Удаление дескриптора не делает недействительными его предков и потомков (в отличие от отзыва дескриптора, который делает недействительными его потомков). То есть через предков и потомков удаленного дескриптора обеспечивается доступ к ресурсу, который они идентифицируют. Также удаление дескриптора не нарушает целостность дерева наследования дескрипторов, которое относится к ресурсу, идентифицируемому этим дескриптором. Место удаленного дескриптора занимает его предок. То есть предок удаленного дескриптора становится непосредственным предком потомков удаленного дескриптора.

Удаление дескрипторов выполняется функцией `KnHandleClose()`, которая объявлена в заголовочном файле `coresrv/handle/handle_api.h`.

`handle_api.h (фрагмент)`

```
/**
 * Функция удаляет дескриптор handle.
 * В случае успеха функция возвращает rcOk, иначе возвращает код ошибки.
 */
Retcode KnHandleClose(Handle handle);
```

Пример использования OCap

В этой статье приведен сценарий использования OCap, в котором программа-сервер предоставляет следующие методы доступа к своим ресурсам:

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

Программа-клиент использует эти методы.

IDL-описание интерфейсных методов:

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

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

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

    b. Поставщик ресурса, 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 ресурса клиенту, в результате которой клиент получает потомка дескриптора, которым владеет поставщик ресурса.

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

3. Клиент вызывает метод использования ресурса `UseResource()`.

    a. Дескриптор, который получен от поставщика ресурса на шаге 2, используется в качестве аргумента метода `UseResource()`. Перед вызовом этого метода клиент упаковывает дескриптор в транспортный

контейнер дескриптора макросом `nk_handle_desc()`.

b. Выполняется разыменование дескриптора, в результате которого поставщик ресурса получает указатель на контекст передачи ресурса.

c. Поставщик ресурса, используя функцию `nk_is_handle_dereferenced()`, проверяет, что выполнена операция разыменования, а не передача дескриптора.

d. Поставщик ресурса проверяет, что права доступа разыменованного дескриптора (который отправлен клиентом) разрешают запрашиваемую операцию над ресурсом, и извлекает указатель на контекст передачи ресурса из транспортного контейнера дескриптора. Для этого поставщик ресурса 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 ресурса, используя контекст передачи ресурса и контекст пользовательского ресурса, выполняет запрашиваемую клиентом операцию над ресурсом. Затем поставщик ресурса отправляет клиенту результат выполнения этой операции.

f. Вызов метода `UseResource()` завершается успешно. Клиент получает результат выполнения операции над ресурсом.

4. Клиент вызывает метод закрытия доступа к ресурсу `CloseResource()`.

a. Дескриптор, который получен от поставщика ресурса на шаге 2, используется в качестве аргумента метода `CloseResource()`. Перед вызовом этого метода клиент упаковывает дескриптор в транспортный контейнер дескриптора макросом `nk_handle_desc()`. После вызова метода `CloseResource()` клиент удаляет дескриптор функцией `KnHandleClose()`.

b. Выполняется разыменование дескриптора, в результате которого поставщик ресурса получает указатель на контекст передачи ресурса.

c. Поставщик ресурса, используя функцию `nk_is_handle_dereferenced()`, проверяет, что выполнена операция разыменования, а не передача дескриптора.

d. Поставщик ресурса, используя функцию `nk_get_badge()`, извлекает указатель на контекст передачи ресурса из транспортного контейнера дескриптора.

e. Поставщик ресурса отзывает дескриптор, которым владеет клиент, функцией `KnHandleRevokeSubtree()`. В качестве аргументов этой функции используются дескриптор ресурса, которым владеет поставщик ресурса, и дескриптор объекта контекста передачи ресурса. Поставщик ресурса получает доступ к этим дескрипторам через указатель на контекст передачи ресурса. (Технически не требуется отзывать дескриптор, которым владеет клиент, так как клиент его уже удалил. Но поставщик ресурса не может быть уверен в том, что клиент удалил дескриптор, поэтому выполняется отзыв).

f. Вызов метода `CloseResource()` завершается успешно.

5. Поставщик ресурса освобождает память, которая была выделена под контекст передачи ресурса и контекст пользовательского ресурса.

a. Поставщик ресурса вызовом функции `KnNoticeGetEvent()` получает уведомление, что объект контекста передачи ресурса закрыт, и удаляет дескриптор объекта контекста передачи ресурса функцией `KnHandleClose()`.

b. Поставщик ресурса вызовом функции `KnNoticeGetEvent()` получает уведомление, что объект контекста передачи ресурса прекратил свое существование, и освобождает память, которая была

выделена под контекст передачи ресурса.

c. Поставщик ресурса удаляет дескриптор ресурса функцией `KnHandleClose()` и освобождает память, которая была выделена под контекст пользовательского ресурса.

Уведомления

Маска 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 (флаги этих событий определены в заголовочном файле `handle/event_descr.h`). Например, в общей части находится флаг `EVENT_OBJECT_DESTROYED`, который определяет событие "прекращение существования ресурса". Специальная часть описывает события, специфичные для пользовательского ресурса. Структура специальной части определяется поставщиком ресурса с использованием макроса `OBJECT_EVENT_SPEC()`, который определен в заголовочном файле `handle/event_descr.h`. Поставщику ресурса необходимо экспортировать публичные заголовочные файлы с описанием структуры специальной части.

EventDesc

Структура, описывающая уведомление, объявлена в файле `coresrv/handle/notice_api.h`.

```
typedef struct {
    rtl_uintptr_t   eventId;
    rtl_uint32_t    eventMask;
} EventDesc;
```

`eventId` – идентификатор записи "ресурс – маска событий" в приемнике уведомлений.

`eventMask` – маска событий, которые произошли.

KnNoticeCreate()

Функция объявлена в файле `coresrv/handle/notice_api.h`.

```
Retcode KnNoticeCreate(Notice *notice);
```

Функция создает приемник уведомлений `notice` (объект, в котором накапливаются уведомления).

В случае успеха функция возвращает **rcOk**, иначе возвращает код ошибки.

KnNoticeGetEvent()

Функция объявлена в файле `coresrv/handle/notice_api.h`.

```
Retcode KnNoticeGetEvent(Notice notice,
                         rtl_uint64_t msec,
                         rtl_size_t countMax,
                         EventDesc *events,
                         rtl_size_t *count);
```

Функция извлекает уведомления из приемника уведомлений `notice`, ожидая наступления событий в течение `msec` миллисекунд.

Входной параметр `countMax` задает максимальное число уведомлений, которое может быть извлечено.

Выходной параметр `events` содержит набор извлеченных уведомлений типа `EventDesc`.

Выходной параметр `count` содержит число уведомлений, которые были извлечены.

В случае успеха функция возвращает **rcOk**, иначе возвращает код ошибки.

Пример

```
const int  maxEventsPerNoticeCall = 10;
Retcode    rc;
EventDesc  events[maxEventsPerNoticeCall];
rtl_size_t eventCount;

rc = KnNoticeGetEvent(notice, INFINITE_TIMEOUT, rtl_countof(events),
                      &events[0], &eventCount);
```

KnNoticeSetObjectEvent()

Функция объявлена в файле `coresrv/handle/notice_api.h`.

```
Retcode KnNoticeSetObjectEvent(Handle object, rtl_uint32_t evMask);
```

Функция сигнализирует, что события из маски событий `evMask` произошли с ресурсом `object`.

Нельзя устанавливать флаги общей части маски событий, так как о событиях из общей части маски событий может сигнализировать только ядро KasperskyOS.

В случае успеха функция возвращает **rcOk**, иначе возвращает код ошибки.

KnNoticeSubscribeToObject()

Функция объявлена в файле `coresrv/handle/notice_api.h`.

```
Retcode KnNoticeSubscribeToObject(Notice notice,
                                  Handle object,
```

```
                                rtl_uint32_t evMask,
                                rtl_uintptr_t evId);
```

Функция добавляет запись вида "ресурс – маска событий" в приемник уведомлений `notice`, чтобы он принимал уведомления о событиях, которые происходят с ресурсом `object` и соответствуют маске событий `evMask`.

Входной параметр `evId` задает идентификатор записи, который назначается пользователем и используется, чтобы идентифицировать запись в полученных уведомлениях.

В случае успеха функция возвращает **rcOk**, иначе возвращает код ошибки.

Пример использования – см. `KnHandleCreateUserObject()`.

Процессы

EntityConnect()

Функция объявлена в заголовочном файле `coresrv/entity/entity_api.h`.

```
Retcode EntityConnect(Entity *cl, Entity *sr);
```

Функция соединяет процессы IPC-каналом. Для этого функция создает IPC-дескрипторы для процесса-клиента `cl` и процесса-сервера `sr`, а затем связывает дескрипторы друг с другом. Создаваемый канал будет включен в группу каналов по умолчанию (имя этой группы совпадает с именем процесса-сервера). Соединяемые процессы должны находиться в остановленном состоянии.

В случае успеха функция возвращает **rcOk**.

EntityConnectToService()

Функция объявлена в заголовочном файле `coresrv/entity/entity_api.h`.

```
Retcode EntityConnectToService(Entity *cl, Entity *sr, const char *name);
```

Функция соединяет процессы IPC-каналом. Для этого функция создает IPC-дескрипторы для процесса-клиента `cl` и процесса-сервера `sr`, а затем связывает дескрипторы друг с другом. Создаваемый канал будет включен в группу каналов с именем `name`. Соединяемые процессы должны находиться в остановленном состоянии.

В случае успеха функция возвращает **rcOk**.

EntityInfo

Структура `EntityInfo`, описывающая процесс, объявлена в файле `if_connection.h`.

```
typedef struct EntityInfo {
    /* имя класса процесса */
    const char            *eiid;
    /* максимальное число служб */
    nk_iid_t              max_endpoints;
    /* информация о службах процесса */
    const EndpointInfo    *endpoints;
    /* аргументы для передачи процессу при его запуске */
    const char            *args[ENTITY_ARGS_MAX + 1];
    /* переменные окружения для передачи процессу при его запуске */
    const char            *envs[ENTITY_ENV_MAX + 1];
    /* флаги процесса */
    EntityFlags           flags;
    /* дерево компонентов процесса */
    const struct nk_component_node *componentTree;
} EntityInfo;

typedef struct EndpointInfo {
    char     *name;       /* полное квалифицированное имя службы */
    nk_iid_t  riid;       /* идентификатор службы */
    char     *iface_name; /* имя интерфейса, который реализует служба */
} EndpointInfo;
typedef enum {
    ENTITY_FLAGS_NONE = 0,
    /* процесс сбрасывается при возникновении необработанного исключения */
    ENTITY_FLAG_DUMPABLE = 1,
} EntityFlags;
```

EntityInit()

Функция объявлена в заголовочном файле `coresrv/entity/entity_api.h`.

```
Entity *EntityInit(const EntityInfo *info);
```

Функция создает процесс. Параметр `info` задает имя класса процесса и (опционально) его службы, аргументы и переменные окружения.

Создаваемый процесс будет иметь имя по умолчанию (совпадает с именем класса процесса), а также и имя исполняемого файлу по умолчанию (также совпадает с именем класса процесса).

В случае успеха функция возвращает структуру, описывающую новый процесс. Созданный процесс находится в остановленном состоянии.

В случае ошибки функция возвращает **RTL_NULL**.

EntityInitEx()

Функция объявлена в заголовочном файле `coresrv/entity/entity_api.h`.

```
Entity *EntityInitEx(const EntityInfo *info, const char *name,
```

```
                        const char *path);
```

Функция создает процесс.

Параметр `info` задает имя класса процесса и (опционально) его службы, аргументы и переменные окружения.

Параметр `name` задает имя процесса. Если он имеет значение **RTL_NULL**, то в качестве имени процесса будет использоваться имя класса процесса из параметра `info`.

Параметр `path` задает имя исполняемого файла в ROMFS-образе решения. Если он имеет значение **RTL_NULL**, то в качестве имени файла будет использоваться имя класса процесса из параметра `info`.

В случае успеха функция возвращает структуру, описывающую новый процесс. Созданный процесс находится в остановленном состоянии.

В случае ошибки функция возвращает **RTL_NULL**.

EntityRun()

Функция объявлена в заголовочном файле `coresrv/entity/entity_api.h`.

```
Retcode EntityRun(Entity *entity);
```

Функция запускает процесс, находящийся в остановленном состоянии. Процесс описывается структурой `entity`.

В случае успеха функция возвращает **rcOk**.

Динамическое создание каналов

KnCmAccept()

Функция объявлена в файле `coresrv/cm/cm_api.h`.

```
Retcode KnCmAccept(const char *client, const char *service, rtl_uint32_t rsid,
                   Handle listener, Handle *handle);
```

Функция принимает запрос клиентского процесса на создание канала, полученный ранее с помощью вызова `KnCmListen()`. Функция вызывается серверным процессом.

Входные параметры:

- `client` – имя клиентского процесса, отправляющего запрос на создание канала;

- `service` – полное квалифицированное имя службы, запрошенное клиентским процессом (например, `blkdev.ata`);
- `rsid` – идентификатор службы;
- `listener` – слушающий дескриптор; если он имеет значение **INVALID_HANDLE**, создается новый слушающий дескриптор, который будет использоваться в качестве серверного IPC-дескриптора создаваемого канала.

Выходной параметр `handle` содержит серверный IPC-дескриптор создаваемого канала.

В случае успеха функция возвращает **rcOk**, иначе возвращает код ошибки.

KnCmConnect()

Функция объявлена в файле `coresrv/cm/cm_api.h`.

```
Retcode KnCmConnect(const char *server, const char *service,
                    rtl_uint32_t msecs, Handle *handle,
                    rtl_uint32_t *rsid);
```

Функция отправляет запрос на создание канала с серверным процессом. Функция вызывается клиентским процессом.

Входные параметры:

- `server` – имя серверного процесса, предоставляющего службу;
- `service` – полное квалифицированное имя службы (например, `blkdev.ata`);
- `msecs` – время ожидания принятия запроса в миллисекундах.

Выходные параметры:

- `handle` – клиентский IPC-дескриптор;
- `rsid` – идентификатор службы.

В случае успеха функция возвращает **rcOk**, иначе возвращает код ошибки.

KnCmDrop()

Функция объявлена в файле `coresrv/cm/cm_api.h`.

```
Retcode KnCmDrop(const char *client, const char *service);
```

Функция отклоняет запрос клиентского процесса на создание канала, полученный ранее с помощью вызова KnCmListen(). Функция вызывается серверным процессом.

Параметры:

- `client` – имя клиентского процесса, отправляющего запрос на создание канала;
- `service` – полное квалифицированное имя службы, запрошенное клиентским процессом (например, `blkdev.ata`).

В случае успеха функция возвращает **rcOk**, иначе возвращает код ошибки.

KnCmListen()

Функция объявлена в файле `coresrv/cm/cm_api.h`.

```
Retcode KnCmListen(const char *filter, rtl_uint32_t msecs, char *client,
                   char *service);
```

Функция проверяет наличие запросов клиентских процессов на создание канала. Функция вызывается серверным процессом.

Входные параметры:

- `filter` – параметр не используется;
- `msecs` – время ожидания запроса в миллисекундах.

Выходные параметры:

- `client` – имя клиентского процесса;
- `service` – полное квалифицированное имя службы, запрошенное клиентским процессом (например, `blkdev.ata`).

В случае успеха функция возвращает **rcOk**, иначе возвращает код ошибки.

NsCreate()

Функция объявлена в файле `coresrv/ns/ns_api.h`.

```
Retcode NsCreate(const char *name, rtl_uint32_t msecs, NsHandle *ns);
```

Функция осуществляет попытку подключиться к серверу имен `name` в течение `msecs` миллисекунд. Если параметр `name` имеет значение **RTL_NULL**, функция пытается подключиться к серверу имен `ns` (серверу имен по умолчанию).

Выходной параметр `ns` содержит дескриптор соединения с сервером имен.

В случае успеха функция возвращает **rcOk**, иначе возвращает код ошибки.

NsEnumServices()

Функция объявлена в файле `coresrv/ns/ns_api.h`.

```
Retcode NsEnumServices(NsHandle ns, const char *type, unsigned index,
                       char *server, rtl_size_t serverSize,
                       char *service, rtl_size_t serviceSize);
```

Функция перечисляет службы с заданным интерфейсом, опубликованные на сервере имен.

Входные параметры:

- `ns` – дескриптор соединения с сервером имен, полученный ранее с помощью вызова `NsCreate()`;
- `type` – имя интерфейса, который реализует служба (например, `kl.drivers.Block`);
- `index` – индекс для перечисления служб;
- `serverSize` – максимальный размер буфера для выходного параметра `server` в байтах;
- `serviceSize` – максимальный размер буфера для выходного параметра `service` в байтах.

Выходные параметры:

- `server` – имя серверного процесса, предоставляющего службу (например, `kl.drivers.Ata`);
- `service` – полное квалифицированное имя службы (например, `blkdev.ata`).

Например, получить полный список серверных процессов, предоставляющих службу с интерфейсом `kl.drivers.Block`, можно следующим образом.

```
rc = NsEnumServices(ns, "kl.drivers.Block", 0, outServerName, ServerNameSize,
outServiceName, ServiceNameSize);
rc = NsEnumServices(ns, "kl.drivers.Block", 1, outServerName, ServerNameSize,
outServiceName, ServiceNameSize);
 ...
rc = NsEnumServices(ns, "kl.drivers.Block", N, outServerName, ServerNameSize,
outServiceName, ServiceNameSize);
```

Вызовы функции с инкрементированием индекса продолжаются до тех пор, пока функция не вернет **rcResourceNotFound**.

В случае успеха функция возвращает **rcOk**, иначе возвращает код ошибки.

NsPublishService()

Функция объявлена в файле `coresrv/ns/ns_api.h`.

```
Retcode NsPublishService(NsHandle ns, const char *type, const char *server,
                         const char *service);
```

Функция публикует службу с заданным интерфейсом на сервере имен.

Параметры:

- `ns` – дескриптор соединения с сервером имен, полученный ранее с помощью вызова `NsCreate()`;
- `type` – имя интерфейса, который реализует публикуемая служба (например, `kl.drivers.Block`);
- `server` – имя серверного процесса (например, `kl.drivers.Ata`);
- `service` – полное квалифицированное имя службы (например, `blkdev.ata`).

В случае успеха функция возвращает **rcOk**, иначе возвращает код ошибки.

NsUnPublishService()

Функция объявлена в файле `coresrv/ns/ns_api.h`.

```
Retcode NsUnPublishService( NsHandle ns, const char *type, const char *server,
                            const char *service);
```

Функция снимает с публикации службу на сервере имен.

Параметры:

- `ns` – дескриптор соединения с сервером имен, полученный ранее с помощью вызова `NsCreate()`;
- `type` – имя интерфейса, который реализует публикуемая служба (например, `kl.drivers.Block`);
- `server` – имя серверного процесса (например, `kl.drivers.Ata`);
- `service` – полное квалифицированное имя службы (например, `blkdev.ata`).

В случае успеха функция возвращает **rcOk**, иначе возвращает код ошибки.

Примитивы синхронизации

KosCondvarBroadcast()

Функция объявлена в файле `kos/condvar.h`.

```
void KosCondvarBroadcast(KosCondvar *condvar);
```

Функция пробуждает все потоки из очереди потоков, заблокированных посредством условной переменной `condvar`.

KosCondvarDeinit()

Функция объявлена в файле `kos/condvar.h`.

```
void KosCondvarDeinit(KosCondvar *condvar);
```

Функция деинициализирует условную переменную `condvar`.

KosCondvarInit()

Функция объявлена в файле `kos/condvar.h`.

```
void KosCondvarInit(KosCondvar *condvar);
```

Функция инициализирует условную переменную `condvar`.

KosCondvarSignal()

Функция объявлена в файле `kos/condvar.h`.

```
void KosCondvarSignal(KosCondvar *condvar);
```

Функция пробуждает один поток из очереди потоков, заблокированных посредством условной переменной `condvar`.

KosCondvarWait()

Функция объявлена в файле `kos/condvar.h`.

```
Retcode KosCondvarWait(KosCondvar *condvar, KosMutex *mutex);
```

Функция блокирует исполнение текущего потока посредством условной переменной `condvar`, пока он не будет пробужден с помощью KosCondvarSignal() или KosCondvarBroadcast().

`mutex` – мьютекс, который будет использован для защиты критической секции.

В случае успеха функция возвращает **rcOk**.

KosCondvarWaitTimeout()

Функция объявлена в файле `kos/condvar.h`.

```
Retcode KosCondvarWaitTimeout(KosCondvar *condvar, KosMutex *mutex,
                              rtl_uint32_t mdelay);
```

Функция блокирует исполнение текущего потока посредством условной переменной `condvar`, пока он не будет пробужден с помощью KosCondvarSignal() или KosCondvarBroadcast(). Поток блокируется не более чем на `mdelay` миллисекунд.

- `mutex` – мьютекс, который будет использован для защиты критической секции.

Функция возвращает **rcOk** в случае успеха и **rcTimeout**, если время ожидания истекло.

KosEventDeinit()

Функция объявлена в файле `kos/event.h`.

```
void KosEventDeinit(KosEvent *event);
```

Функция освобождает ресурсы, связанные с событием `event` (уничтожает событие).

KosEventInit()

Функция объявлена в файле `kos/event.h`.

```
void KosEventInit(KosEvent *event);
```

Функция создает событие `event`.

Созданное событие находится в несигнальном состоянии.

KosEventReset()

Функция объявлена в файле `kos/event.h`.

```
void KosEventReset(KosEvent *event);
```

Функция переводит событие `event` в несигнальное состояние (сбрасывает событие).

KosEventSet()

Функция объявлена в файле `kos/event.h`.

```
void KosEventSet(KosEvent *event);
```

Функция переводит событие `event` в сигнальное состояние (сигнализирует событие) и таким образом пробуждает все потоки, ожидающие его.

KosEventWait()

Функция объявлена в файле `kos/event.h`.

```
void KosEventWait(KosEvent *event, rtl_bool reset);
```

Функция ожидает перехода события в сигнальное состояние.

Параметр `reset` указывает, следует ли автоматически сбросить событие при успешном завершении ожидания.

Функция возвращает **rcOk** в случае успеха.

KosEventWaitTimeout()

Функция объявлена в файле `kos/event.h`.

```
Retcode KosEventWaitTimeout(KosEvent *event, rtl_bool reset,
                            rtl_uint32_t msec);
```

Функция ожидает перехода события в сигнальное состояние в течение `msec` миллисекунд.

Параметр `reset` указывает, следует ли автоматически сбросить событие при успешном завершении ожидания.

Функция возвращает **rcOk** в случае успеха и **rcTimeout** при превышении таймаута.

KosMutexDeinit()

Функция объявлена в файле `kos/mutex.h`.

```
void KosMutexDeinit(KosMutex *mutex);
```

Функция уничтожает мьютекст `mutex`.

KosMutexInit()

Функция объявлена в файле `kos/mutex.h`.

```
void KosMutexInit(KosMutex *mutex);
```

Функция выполняет инициализацию мьютекса `mutex` в незаблокированном состоянии.

KosMutexInitEx()

Функция объявлена в файле `kos/mutex.h`.

```
void KosMutexInitEx(KosMutex *mutex, int recursive);
```

Функция выполняет инициализацию мьютекса `mutex` в незаблокированном состоянии.

Для инициализации рекурсивного мьютекса в параметр `recursive` нужно передать значение **1**.

KosMutexLock()

Функция объявлена в файле `kos/mutex.h`.

```
void KosMutexLock(KosMutex *mutex);
```

Функция захватывает мьютекс `mutex`.

Если мьютекс уже захвачен, поток блокируется в ожидании его разблокировки.

KosMutexLockTimeout()

Функция объявлена в файле `kos/mutex.h`.

```
Retcode KosMutexLockTimeout(KosMutex *mutex, rtl_uint32_t mdelay);
```

Функция захватывает мьютекс `mutex`.

Если мьютекс уже захвачен, поток блокируется на `mdelay` миллисекунд в ожидании его разблокировки.

Функция возвращает **rcOk** в случае успеха и **rcTimeout**, если время ожидания истекло.

KosMutexTryLock()

Функция объявлена в файле `kos/mutex.h`.

```
Retcode KosMutexTryLock(KosMutex *mutex);
```

Функция делает попытку захвата мьютекса `mutex`.

Функция возвращает **rcOk**, если мьютекс удалось захватить и **rcBusy**, если мьютекс не удалось захватить, так как он уже захвачен.

KosMutexUnlock()

Функция объявлена в файле `kos/mutex.h`.

```
void KosMutexUnlock(KosMutex *mutex);
```

Функция разблокирует мьютекс `mutex`.

Для разблокировки рекурсивного мьютекса нужно сделать столько вызовов `KosMutexUnlock()`, сколько раз рекурсивный мьютекс был заблокирован.

KosRWLockDeinit()

Функция объявлена в файле `kos/rwlock.h`.

```
void KosRWLockDeinit(KosRWLock *rwlock);
```

Функция деинициализирует блокировку чтения-записи `rwlock`.

KosRWLockInit()

Функция объявлена в файле `kos/rwlock.h`.

```
void KosRWLockInit(KosRWLock *rwlock);
```

Функция инициализирует блокировку чтения-записи `rwlock`.

KosRWLockRead()

Функция объявлена в файле `kos/rwlock.h`.

```
void KosRWLockRead(KosRWLock *rwlock);
```

Функция блокирует потоки чтения.

KosRWLockTryRead()

Функция объявлена в файле `kos/rwlock.h`.

```
Retcode KosRWLockTryRead(KosRWLock *rwlock);
```

Функция делает попытку блокировки потоков чтения.

В случае успеха функция возвращает **rcOk**.

KosRWLockTryWrite()

Функция объявлена в файле `kos/rwlock.h`.

```
Retcode KosRWLockTryWrite(KosRWLock *rwlock);
```

Функция делает попытку блокировки потоков записи.

В случае успеха функция возвращает **rcOk**.

KosRWLockUnlock()

Функция объявлена в файле `kos/rwlock.h`.

```
void KosRWLockUnlock(KosRWLock *rwlock);
```

Функция снимает блокировку чтения-записи `rwlock`.

KosRWLockWrite()

Функция объявлена в файле `kos/rwlock.h`.

```
void KosRWLockWrite(KosRWLock *rwlock);
```

Функция блокирует потоки записи.

KosSemaphoreDeinit()

Функция объявлена в файле `kos/semaphore.h`.

```
Retcode KosSemaphoreDeinit(KosSemaphore *semaphore);
```

Функция уничтожает семафор `semaphore`, инициализированный ранее функцией `KosSemaphoreInit()`.

> Безопасно уничтожать инициализированный семафор, на котором в настоящее время нет заблокированных потоков. Эффект уничтожения семафора, на котором в данный момент заблокированы другие потоки, непредсказуем.

Функция возвращает:

- **rcOk** в случае успеха;
- **rcInvalidArgument**, если `semaphore` указывает на невалидный семафор;
- **rcFail**, если есть потоки, заблокированные этим семафором.

KosSemaphoreInit()

Функция объявлена в файле `kos/semaphore.h`.

```
Retcode KosSemaphoreInit(KosSemaphore *semaphore, unsigned count);
```

Функция инициализирует семафор `semaphore` с начальным значением `count`.

Функция возвращает:

- **rcOk** в случае успеха;
- **rcInvalidArgument**, если `semaphore` указывает на невалидный семафор;
- **rcFail**, если значение `count` превышает **KOS_SEMAPHORE_VALUE_MAX**.

KosSemaphoreSignal()

Функция объявлена в файле `kos/semaphore.h`.

```
Retcode KosSemaphoreSignal(KosSemaphore *semaphore);
```

Функция освобождает (сигнализирует) семафор `semaphore`.

Функция возвращает:

- **rcOk** в случае успеха;
- **rcInvalidArgument**, если `semaphore` указывает на невалидный семафор.

KosSemaphoreTryWait()

Функция объявлена в файле `kos/semaphore.h`.

```
Retcode KosSemaphoreTryWait(KosSemaphore *semaphore);
```

Функция делает попытку захвата семафора `semaphore`.

Функция возвращает:

- **rcOk** в случае успеха;
- **rcInvalidArgument**, если `semaphore` указывает на невалидный семафор;
- **rcBusy**, если семафор уже захвачен.

KosSemaphoreWait()

Функция объявлена в файле `kos/semaphore.h`.

```
Retcode KosSemaphoreWait(KosSemaphore *semaphore);
```

Функция ожидает захвата семафора `semaphore`.

Функция возвращает:

- **rcOk** в случае успеха;
- **rcInvalidArgument**, если `semaphore` указывает на невалидный семафор.

KosSemaphoreWaitTimeout()

Функция объявлена в файле `kos/semaphore.h`.

```
Retcode KosSemaphoreWaitTimeout(KosSemaphore *semaphore, rtl_uint32_t mdelay);
```

Функция ожидает захвата семафора `semaphore` в течение `mdelay` миллисекунд.

Функция возвращает:

- **rcOk** в случае успеха;
- **rcInvalidArgument**, если `semaphore` указывает на невалидный семафор;
- **rcTimeout**, если время ожидания истекло.

DMA-буферы

DmaInfo

Структура, описывающая DMA-буфер, объявлена в файле `io/io_dma.h`.

```
typedef struct {
    /** DMA-флаги (атрибуты). */
    DmaAttr         flags;

    /** Минимальный порядок DMA-блоков в буфере. */
    rtl_size_t      orderMin;

    /** Размер DMA-буфера. */
    rtl_size_t      size;

    /** Число DMA-блоков (меньше или равно DMA_FRAMES_COUNT_MAX).
     *  Может быть равно 0, если DMA-буфер недоступен для устройства. */
    rtl_size_t      count;

    /** Массив описателей DMA-блоков. */
    union   DmaFrameDescriptor {
        struct {
            /** Порядок (order) DMA-блока. Число страниц в блоке равно двум
             *  в степени order. */
            DmaAddr order: DMA_FRAME_ORDER_BITS;

            /** Физический или IOMMU-адрес DMA-блока. */
            DmaAddr frame: DMA_FRAME_BASE_BITS;
        };

        /** Описатель DMA-блока */
        DmaAddr raw;
    } descriptors[1];
} DmaInfo;
```

DMA-флаги

DMA-флаги (атрибуты) объявлены в файле `io/io_dma.h`.

- **DMA_DIR_TO_DEVICE** – разрешены транзакции из основной памяти в память устройства;
- **DMA_DIR_FROM_DEVICE** – разрешены транзакции из памяти устройства в основную память;
- **DMA_DIR_BIDIR** – разрешены транзакции из основной памяти в память устройства и наоборот;
- **DMA_ZONE_DMA32** – под буфер разрешено использовать только первые 4 Гб памяти;
- **DMA_ATTR_WRITE_BACK**, **DMA_ATTR_WRITE_THROUGH**, **DMA_ATTR_CACHE_DISABLE**, **DMA_ATTR_WRITE_COMBINE** – управление кэшированием страниц памяти.

KnIoDmaBegin()

Функция объявлена в файле `coresrv/io/dma.h`.

```
Retcode KnIoDmaBegin(Handle rid, Handle *handle);
```

Функция разрешает устройству доступ к DMA-буферу с дескриптором `rid`.

Выходной параметр `handle` содержит дескриптор данного разрешения.

В случае успеха функция возвращает **rcOk**.

Пример использования – см. KnIoDmaCreate().

> Чтобы запретить устройству доступ к DMA-буферу, необходимо вызвать функцию `KnIoClose()`, передав в нее дескриптор разрешения `handle`.

KnIoDmaCreate()

Функция объявлена в файле `coresrv/io/dma.h`.

```
Retcode KnIoDmaCreate(rtl_uint32_t order, rtl_size_t size, DmaAttr flags,
                      Handle *outRid);
```

Функция регистрирует и выделяет физический DMA-буфер.

Входные параметры:

- `order` – минимальный допустимый порядок выделения DMA-блоков; фактический порядок каждого блока в DMA-буфере выбирается ядром (но не будет меньше order) и помещается в дескриптор блока; порядок блока определяет число страниц в нем: блок с порядком **N** состоит из **2^N** страниц;
- `size` – размер DMA-буфера в байтах (должен быть кратен размеру страницы); сумма размеров выделенных DMA-блоков будет не меньше `size`;
- `flags` – DMA-флаги.

Выходной параметр `outRid` содержит дескриптор выделенного DMA-буфера.

В случае успеха функция возвращает **rcOk**.

> Если DMA-буфер больше не используется, его необходимо освободить с помощью функции `KnIoClose()`.

Пример

```
Retcode RegisterDmaMem(rtl_size_t    size,
                       DmaAttr     attr,
                       Handle      *handle,
                       Handle      *dmaHandle,
                       Handle      *mappingHandle,
                       void        **addr)
{
    Retcode     ret;

    *handle = INVALID_HANDLE;
    *dmaHandle = INVALID_HANDLE;
    *mappingHandle = INVALID_HANDLE;

    ret = KnIoDmaCreate(rtl_roundup_order(size >> PAGE_SHIFT),
                        size,
                        attr,
                        handle);

    if (ret == rcOk) {
        ret = KnIoDmaBegin(*handle, dmaHandle);
    }

    if (ret == rcOk) {
        ret = KnIoDmaMap(*handle,
                         0,
                         size,
                         RTL_NULL,
                         VMM_FLAG_READ | VMM_FLAG_WRITE,
                         addr,
                         mappingHandle);
    }

    if (ret != rcOk) {
        if (*mappingHandle != INVALID_HANDLE)
            KnHandleClose(*mappingHandle);

        if (*dmaHandle != INVALID_HANDLE)
            KnHandleClose(*dmaHandle);

        if (*handle != INVALID_HANDLE)
            KnHandleClose(*handle);
    }

    return ret;
}
```

KnIoDmaGetInfo()

Функция объявлена в файле `coresrv/io/dma.h`.

```
Retcode KnIoDmaGetInfo(Handle rid, DmaInfo **outInfo);
```

Функция получает информацию о DMA-буфере с дескриптором `rid`.

Выходной параметр `outInfo` содержит информацию о DMA-буфере.

В случае успеха функция возвращает **rcOk**.

> В отличие от `KnIoDmaGetPhysInfo()`, параметр `outInfo` содержит не физические, а IOMMU-адреса DMA-блоков.

KnIoDmaGetPhysInfo()

Функция объявлена в файле `coresrv/io/dma.h`.

```
Retcode KnIoDmaGetPhysInfo(Handle rid, DmaInfo **outInfo);
```

Функция получает информацию о DMA-буфере с дескриптором `rid`.

Выходной параметр `outInfo` содержит информацию о DMA-буфере.

В случае успеха функция возвращает **rcOk**.

> В отличие от `KnIoDmaGetInfo()`, параметр `outInfo` содержит не IOMMU-адреса, а физические адреса DMA-блоков.

KnIoDmaMap()

Функция объявлена в файле `coresrv/io/dma.h`.

```
Retcode KnIoDmaMap(Handle rid, rtl_size_t offset, rtl_size_t length, void *hint,
                   int vmflags, void **addr, Handle *handle);
```

Функция отображает участок DMA-буфера на адресное пространство процесса.

Входные параметры:

- `rid` – дескриптор выделенного с помощью `KnIoDmaCreate()` DMA-буфера;
- `offset` – странично-выровненное смещение начала участка от начала буфера в байтах;
- `length` – размер участка; должен быть кратен размеру страницы и не превышать <размер буфера - `offset`>;
- `hint` – виртуальный адрес начала отображения; если он равен **0**, адрес выберет ядро;

- `vmflags` – флаги аллокации.

В параметре `vmflags` можно использовать следующие флаги аллокации (`vmm/flags.h`):

- **VMM_FLAG_READ** и **VMM_FLAG_WRITE** – атрибуты защиты памяти;
- **VMM_FLAG_LOW_GUARD** и **VMM_FLAG_HIGH_GUARD** – добавление защитной страницы перед и после выделенной памяти соответственно.

Допустимые комбинации атрибутов защиты памяти:

- **VMM_FLAG_READ** – разрешено чтение содержимого страницы;
- **VMM_FLAG_WRITE** – разрешено изменение содержимого страницы;
- **VMM_FLAG_READ | VMM_FLAG_WRITE** – разрешено чтение и изменение содержимого страницы.

Выходные параметры:

- `addr` – указатель на виртуальный адрес начала отображенного участка;
- `handle` – дескриптор созданного отображения.

В случае успеха функция возвращает **rcOk**.

Пример использования – см. KnIoDmaCreate().

> Чтобы удалить созданное отображение, необходимо вызвать функцию `KnIoClose()`, передав в нее дескриптор отображения `handle`.

IOMMU

KnIommuAttachDevice()

Функция объявлена в файле `coresrv/iommu/iommu_api.h`.

```
Retcode KnIommuAttachDevice(rtl_uint16_t bdf);
```

Функция добавляет PCI-устройство с идентификатором `bdf` в группу IOMMU вызывающего процесса (IOMMU domain).

В случае успеха возвращает **rcOk**.

KnIommuDetachDevice()

Функция объявлена в файле `coresrv/iommu/iommu_api.h`.

```
Retcode KnIommuDetachDevice(rtl_uint16_t bdf);
```

Функция удаляет PCI-устройство с идентификатором `bdf` из группы IOMMU вызывающего процесса (IOMMU domain).

В случае успеха функция возвращает **rcOk**.

Порты ввода-вывода

IoReadIoPort8(), IoReadIoPort16(), IoReadIoPort32()

Функции объявлены в файле `coresrv/io/ports.h`.

```
rtl_uint8_t IoReadIoPort8(rtl_size_t port);
rtl_uint16_t IoReadIoPort16(rtl_size_t port);
rtl_uint32_t IoReadIoPort32(rtl_size_t port);
```

Функции вычитывают один, два или четыре байта соответственно из порта `port` и возвращают прочитанное значение.

IoReadIoPortBuffer8(), IoReadIoPortBuffer16(), IoReadIoPortBuffer32()

Функции объявлены в файле `coresrv/io/ports.h`.

```
void IoReadIoPortBuffer8(rtl_size_t port, rtl_uint8_t *dst, rtl_size_t cnt);
void IoReadIoPortBuffer16(rtl_size_t port, rtl_uint16_t *dst, rtl_size_t cnt);
void IoReadIoPortBuffer32(rtl_size_t port, rtl_uint32_t *dst, rtl_size_t cnt);
```

Функции вычитывают последовательность одно-, двух- или четырехбайтовых значений соответственно из порта `port` и записывают значения в массив `dst`.

`cnt` – длина последовательности.

IoWriteIoPort8(), IoWriteIoPort16(), IoWriteIoPort32()

Функции объявлены в файле `coresrv/io/ports.h`.

```
void IoWriteIoPort8(rtl_size_t port, rtl_uint8_t data);
void IoWriteIoPort16(rtl_size_t port, rtl_uint16_t data);
void IoWriteIoPort32(rtl_size_t port, rtl_uint32_t data);
```

Функции записывают одно-, двух- или четырехбайтовое значение `data` в порт `port`.

IoWriteIoPortBuffer8(), IoWriteIoPortBuffer16(), IoWriteIoPortBuffer32()

Функции объявлены в файле `coresrv/io/ports.h`.

```
void IoWriteIoPortBuffer8(rtl_size_t port, const rtl_uint8_t *src,
                          rtl_size_t cnt);
void IoWriteIoPortBuffer16(rtl_size_t port, const rtl_uint16_t *src,
                           rtl_size_t cnt);
void IoWriteIoPortBuffer32(rtl_size_t port, const rtl_uint32_t *src,
                           rtl_size_t cnt);
```

Функции записывают последовательность одно-, двух- или четырехбайтовых значений соответственно из массива `src` в порт `port`.

`cnt` – длина последовательности.

KnIoPermitPort()

Функция объявлена в файле `coresrv/io/ports.h`.

```
Retcode KnIoPermitPort(Handle rid, Handle *handle);
```

Функция разрешает процессу доступ к порту (диапазону портов) с дескриптором `rid`.

Выходной параметр `handle` содержит дескриптор данного разрешения.

Функция возвращает **rcOk** в случае успеха.

Пример

```
static Retcode PortInit(IOPort *resource)
{
    Retcode  rc = rcFail;
        rc = KnRegisterPorts(resource->base,
                             resource->size,
                             &resource->handle);
        if (rc == rcOk)
            rc = KnIoPermitPort(resource->handle, &resource->permitHandle);
        resource->addr = (void *) (rtl_uintptr_t) resource->base;

    return rc;
}
```

KnRegisterPort8(), KnRegisterPort16(), KnRegisterPort32()

Функции объявлены в файле `coresrv/io/ports.h`.

```
Retcode KnRegisterPort8(rtl_uint16_t port, Handle *outRid);
Retcode KnRegisterPort16(rtl_uint16_t port, Handle *outRid);
Retcode KnRegisterPort32(rtl_uint16_t port, Handle *outRid);
```

Функции регистрируют восьми-, шестнадцати- или тридцатидвухбитный порт соответственно с адресом `port` и назначают ему дескриптор `outRid`.

Функции возвращают **rcOk** в случае успешного выделения порта.

> Если порт больше не используется, его необходимо освободить с помощью функции `KnIoClose()`.

KnRegisterPorts()

Функция объявлена в файле `coresrv/io/ports.h`.

```
Retcode KnRegisterPorts(rtl_uint16_t port, rtl_size_t size, Handle *outRid);
```

Функция регистрирует диапазон портов (участок памяти) с базовым адресом `port` и размером `size` в байтах и назначает ему дескриптор `outRid`.

Возвращает **rcOk** в случае успешного выделения диапазона портов.

Пример использования – см. `KnIoPermitPort()`.

> Если диапазон портов больше не используется, его необходимо освободить с помощью функции `KnIoClose()`.

Ввод-вывод через память (MMIO)

IoReadMmBuffer8(), IoReadMmBuffer16(), IoReadMmBuffer32()

Функции объявлены в файле `coresrv/io/mmio.h`.

```
void IoReadMmBuffer8(volatile rtl_uint8_t *baseReg, rtl_uint8_t *dst,
                     rtl_size_t cnt);
void IoReadMmBuffer16(volatile rtl_uint16_t *baseReg, rtl_uint16_t *dst,
                      rtl_size_t cnt);
void IoReadMmBuffer32(volatile rtl_uint32_t *baseReg, rtl_uint32_t *dst,
                      rtl_size_t cnt);
```

Функции вычитывают последовательность одно-, двух- или четырехбайтовых значений соответственно из регистра, отображаемого по адресу `baseReg`, и записывают значения в массив `dst`. `cnt` – длина последовательности.

IoReadMmReg8(), IoReadMmReg16(), IoReadMmReg32()

Функции объявлены в файле `coresrv/io/mmio.h`.

```
rtl_uint8_t IoReadMmReg8(volatile void *reg);
rtl_uint16_t IoReadMmReg16(volatile void *reg);
rtl_uint32_t IoReadMmReg32(volatile void *reg);
```

Функции вычитывают один, два или четыре байта соответственно из регистра, отображаемого по адресу `reg`, и возвращают прочитанное значение.

IoWriteMmBuffer8(), IoWriteMmBuffer16(), IoWriteMmBuffer32()

Функции объявлены в файле `coresrv/io/mmio.h`.

```
void IoWriteMmBuffer8(volatile rtl_uint8_t *baseReg, const rtl_uint8_t *src,
                      rtl_size_t cnt);
void IoWriteMmBuffer16(volatile rtl_uint16_t *baseReg, const rtl_uint16_t *src,
                       rtl_size_t cnt);
void IoWriteMmBuffer32(volatile rtl_uint32_t *baseReg, const rtl_uint32_t *src,
                       rtl_size_t cnt);
```

Функции записывают последовательность одно-, двух- или четырехбайтовых значений соответственно из массива src в регистр, отображаемый по адресу `baseReg`. `cnt` - длина последовательности.

IoWriteMmReg8(), IoWriteMmReg16(), IoWriteMmReg32()

Функции объявлены в файле `coresrv/io/mmio.h`.

```
void IoWriteMmReg8(volatile void *reg, rtl_uint8_t data);
void IoWriteMmReg16(volatile void *reg, rtl_uint16_t data);
void IoWriteMmReg32(volatile void *reg, rtl_uint32_t data);
```

Функции записывают одно-, двух- или четырехбайтовое значение `data` в регистр, отображаемый по адресу `reg`.

KnIoMapMem()

Функция объявлена в файле `coresrv/io/mmio.h`.

```
Retcode KnIoMapMem(Handle rid, rtl_uint32_t prot, rtl_uint32_t attr,
                   void **addr, Handle *handle);
```

Функция отображает зарегистрированный участок памяти, которому назначен дескриптор `rid`, на адресное пространство процесса.

С помощью входных параметров `prot` и `attr` можно изменить атрибуты защиты участка памяти, а также отключить кэширование.

Выходные параметры:

- `addr` – указатель на адрес начала участка виртуальной памяти;
- `handle` – дескриптор участка виртуальной памяти.

Функция возвращает **rcOk** в случае успеха.

`prot` – атрибуты защиты участка памяти через MMU, возможные значения:

- **VMM_FLAG_READ** – разрешено чтение;
- **VMM_FLAG_WRITE** – разрешена запись;
- **VMM_FLAG_READ | VMM_FLAG_WRITE** – разрешены чтение и запись;
- **VMM_FLAG_RWX_MASK** или **VMM_FLAG_READ | VMM_FLAG_WRITE | VMM_FLAG_EXECUTE** – полный доступ к участку памяти (эти записи эквивалентны).

`attr` – атрибуты участка памяти, возможные значения:

- **VMM_FLAG_CACHE_DISABLE** – отключить кэширование;
- **VMM_FLAG_LOW_GUARD** и **VMM_FLAG_HIGH_GUARD** – добавление защитной страницы перед и после выделенной памяти соответственно;
- **VMM_FLAG_ALIAS** – флаг указывает, что участок памяти может иметь несколько виртуальных адресов.

Пример

```
static Retcode MemInit(IOMem *resource)
{
    Retcode  rc = rcFail;
        rc = KnRegisterPhyMem(resource->base,
                              resource->size,
                              &resource->handle);

        if (rc == rcOk)
            rc = KnIoMapMem(resource->handle,
                            VMM_FLAG_READ | VMM_FLAG_WRITE,
                            VMM_FLAG_CACHE_DISABLE,
                            (void **) &resource->addr, &resource->permitHandle);

        if (rc == rcOk)
            resource->addr = ((rtl_uint8_t *) resource->addr
                              + resource->offset);
```

```
    return rc;
}
```

KnRegisterPhyMem()

Функция объявлена в файле `coresrv/io/mmio.h`.

```
Retcode KnRegisterPhyMem(rtl_uint64_t addr, rtl_size_t size, Handle *outRid);
```

Функция регистрирует участок памяти размером `size` байт с началом по адресу `addr`.

В случае успешной регистрации дескриптор, назначенный участку памяти, будет передан в параметр `outRid`, а функция вернет **rcOk**.

Адрес `addr` должен быть странично-выровненным, а размер `size` должен быть кратен размеру страницы.

Пример использования – см. `KnIoMapMem()`.

> Если участок памяти больше не используется, его необходимо освободить с помощью функции `KnIoClose()`.

Прерывания

> Описанный здесь интерфейс является низкоуровневым. Для работы с прерываниями в большинстве случаев рекомендуется использовать интерфейс, предоставляемый библиотекой kdf.

KnIoAttachIrq()

Функция объявлена в файле `coresrv/io/irq.h`.

```
Retcode KnIoAttachIrq(Handle rid, rtl_uint32_t flags, Handle *handle);
```

Функция привязывает вызывающий поток к прерыванию.

Входные параметры:

- `rid` – дескриптор прерывания, полученный с помощью вызова `KnRegisterIrq()`;
- `flags` – флаги прерывания.

Выходной параметр `handle` содержит IPC-дескриптор, посредством которого вызывающий поток будет ожидать прерывание, выполнив вызов `Recv()`.

В случае успеха функция возвращает **rcOk**, иначе возвращает код ошибки.

Флаги прерывания

- **IRQ_LEVEL_LOW** – генерация по низкому уровню;
- **IRQ_LEVEL_HIGH** – генерация по высокому уровню;
- **IRQ_EDGE_RAISE** – генерация по переднему фронту;
- **IRQ_EDGE_FALL** – генерация по заднему фронту;
- **IRQ_SHARED** – разделяемое прерывание;
- **IRQ_PRIO_LOW** – низкий приоритет прерывания;
- **IRQ_PRIO_NORMAL** – нормальный приоритет;
- **IRQ_PRIO_HIGH** – высокий приоритет;
- **IRQ_PRIO_RT** – приоритет реального времени.

KnIoDetachIrq()

Функция объявлена в файле `coresrv/io/irq.h`.

```
Retcode KnIoDetachIrq(Handle rid);
```

Функция отвязывает вызывающий поток от прерывания.

`rid` – дескриптор прерывания, полученный с помощью вызова `KnRegisterIrq()`;

В случае успеха функция возвращает **rcOk**, иначе возвращает код ошибки.

KnIoDisableIrq()

Функция объявлена в файле `coresrv/io/irq.h`.

```
Retcode KnIoDisableIrq(Handle rid);
```

Функция маскирует (запрещает) прерывание с дескриптором `rid`.

В случае успеха функция возвращает **rcOk**.

KnIoEnableIrq()

Функция объявлена в файле `coresrv/io/irq.h`.

```
Retcode KnIoEnableIrq(Handle rid);
```

Функция демаскирует (разрешает) прерывание с дескриптором `rid`.

В случае успеха функция возвращает **rcOk**.

KnRegisterIrq()

Функция объявлена в файле `coresrv/io/irq.h`.

```
Retcode KnRegisterIrq(int irq, Handle *outRid);
```

Функция регистрирует прерывание с номером `irq`.

Выходной параметр `outRid` содержит дескриптор прерывания.

В случае успеха функция возвращает **rcOk**.

> Если прерывание больше не используется, его необходимо освободить с помощью функции `KnIoClose()`.

Освобождение ресурсов

KnIoClose()

Функция объявлена в файле `coresrv/io/io_api.h`.

```
Retcode KnIoClose(Handle rid);
```

Функция освобождает зарегистрированный ресурс ввода-вывода (порт/порты ввода-вывода, DMA-буфер, прерывание или участок памяти для MMIO) с дескриптором `rid`.

В случае успешного освобождения функция возвращает **rcOk**.

Пример использования – см. KnIoDmaCreate().

Время

KnGetMSecSinceStart()

Функция объявлена в файле `coresrv/time/time_api.h`.

```
rtl_size_t KnGetMSecSinceStart(void);
```

Функция возвращает количество миллисекунд, прошедших с момента старта системы.

KnGetRtcTime()

Функция объявлена в файле `coresrv/time/time_api.h`.

```
Retcode KnGetRtcTime(RtlRtcTime *rt);
```

Функция записывает в параметр `rt` системное POSIX-время в формате RTC.

В случае успеха возвращает **rcOk**, в случае ошибки – **rcFail**.

Формат времени RTC задается структурой `RtlRtcTime` (объявлена в файле `rtl/rtc.h`.):

```
typedef struct {
    rtl_uint32_t msec;  /**< миллисекунды                */
    rtl_uint32_t sec;   /**< секунда (0..59)             */
    rtl_uint32_t min;   /**< минута (0..59)              */
    rtl_uint32_t hour;  /**< час (0..23)                 */
    rtl_uint32_t mday;  /**< день (1..31)                */
    rtl_uint32_t month; /**< месяц (0..11)               */
    rtl_int32_t year;   /**< год - 1900                  */
    rtl_uint32_t wday;  /**< день недели (0..6)          */
} RtlRtcTime;
```

KnGetSystemTime()

Функция объявлена в файле `coresrv/time/time_api.h`.

```
Retcode KnGetSystemTime(RtlTimeSpec *time);
```

Функция позволяет получить системное время.

Выходной параметр `time` содержит системное POSIX-время в формате `RtlTimeSpec`.

KnSetSystemTime()

Функция объявлена в файле `coresrv/time/time_api.h`.

```
Retcode KnSetSystemTime(RtlTimeSpec *time);
```

Функция позволяет задать системное время.

Параметр `time` должен содержать POSIX-время в формате `RtlTimeSpec`.

Не рекомендуется вызывать функцию `KnSetSystemTime()` в потоке обработчика прерываний.

KnGetSystemTimeRes()

Функция объявлена в файле `coresrv/time/time_api.h`.

```
Retcode KnGetSystemTimeRes(RtlTimeSpec *res);
```

Функция позволяет получить разрешение источника системного времени.

Выходной параметр `res` содержит разрешение формате `RtlTimeSpec`.

KnGetUpTime()

Функция объявлена в файле `coresrv/time/time_api.h`.

```
Retcode KnGetUpTime(RtlTimeSpec *time);
```

Функция позволяет получить время, прошедшее с момента старта системы.

Выходной параметр `time` содержит время в формате `RtlTimeSpec`.

KnGetUpTimeRes()

Функция объявлена в файле `coresrv/time/time_api.h`.

```
Retcode KnGetUpTimeRes(RtlTimeSpec *res);
```

Функция позволяет получить разрешение источника времени, значение которого можно получить через `KnGetUpTime()`.

Выходной параметр `res` содержит разрешение в формате `RtlTimeSpec`.

RtlTimeSpec

Формат времени timespec задается структурой `RtlTimeSpec` (объявлена в файле `rtl/rtc.h`).

```
typedef struct {
    rtl_time_t   sec;   /**< целое число секунд, прошедшее с начала Unix-эпохи
                         *   или другого заданного момента времени */
    rtl_nsecs_t  nsec;  /**< поправка в наносекундах (число наносекунд,
                         *   прошедших с момента, заданного числом секунд*/
} RtlTimeSpec;
```

Очереди

KosQueueAlloc()

Функция объявлена в файле `kos/queue.h`.

```
void *KosQueueAlloc(KosQueueHandle queue);
```

Функция аллоцирует память под новый объект из буфера очереди `queue`.

В случае успеха функция возвращает указатель на память под объект, если буфер заполнен – **RTL_NULL**.

KosQueueCreate()

Функция объявлена в файле `kos/queue.h`.

```
KosQueueHandle KosQueueCreate(unsigned objCount,
                              unsigned objSize,
                              unsigned objAlign,
                              void *buffer);
```

Функция создает очередь объектов (fifo) и связанный с ней буфер.

Параметры:

- `objCount` – максимальное количество объектов в очереди;
- `objSize` – размер объекта (байт);
- `objAlign` – выравнивание объекта в байтах, должно быть степенью двойки;

- `buffer` – указатель на внешний буфер под объекты; если задать его равным **RTL_NULL**, то буфер будет выделен с помощью функции KosMemAlloc().

Функция возвращает дескриптор созданной очереди и **RTL_NULL** в случае ошибки.

KosQueueDestroy()

Функция объявлена в файле `kos/queue.h`.

```
void KosQueueDestroy(KosQueueHandle queue);
```

Функция удаляет очередь `queue` и освобождает выделенный под нее буфер.

KosQueueFlush()

Функция объявлена в файле `kos/queue.h`.

```
void KosQueueFlush(KosQueueHandle queue);
```

Функция извлекает все объекты из очереди `queue` и освобождает всю память, занятую ими.

KosQueueFree()

Функция объявлена в файле `kos/queue.h`.

```
void KosQueueFree(KosQueueHandle queue, void *obj);
```

Функция освобождает память, занимаемую объектом `obj` в буфере очереди `queue`.

Указатель `obj` может быть получен вызовом функции KosQueueAlloc() или KosQueuePop().

Пример использования – см. KosQueuePop().

KosQueuePop()

Функция объявлена в файле `kos/queue.h`.

```
void *KosQueuePop(KosQueueHandle queue, rtl_uint32_t timeout);
```

Функция извлекает из начала очереди `queue` объект и возвращает указатель на него.

Параметр `timeout` определяет поведение функции в случае, если очередь пуста:

- **0** – немедленный возврат **RTL_NULL**;
- **INFINITE_TIMEOUT** – блокировка в ожидании нового объекта в очереди;
- любое другое значение `timeout` – ожидание нового объекта в очереди в течение `timeout` миллисекунд; по истечении этого времени возвращается **RTL_NULL**.

Пример

```
int GpioEventDispatch(void *context)
{
    GpioEvent *event;
    GpioDevice *device = context;
    rtl_bool proceed = rtl_true;

    do {
        event = KosQueuePop(device->queue, INFINITE_TIMEOUT);
        if (event != RTL_NULL) {

            if (event->type == GPIO_EVENT_TYPE_THREAD_ABORT) {
                proceed = rtl_false;
            } else {
                GpioDeliverEvent(device, event);
            }
            KosQueueFree(device->queue, event);
        }
    } while (proceed);

    KosPutObject(device);
    return rcOk;
}
```

KosQueuePush()

Функция объявлена в файле `kos/queue.h`.

```
void KosQueuePush(KosQueueHandle queue, void *obj);
```

Функция добавляет объект `obj` в конец очереди `queue`.

Указатель `obj` может быть получен вызовом функции `KosQueueAlloc()` или `KosQueuePop()`.

Барьеры памяти

IoReadBarrier()

Функция объявлена в файле `coresrv/io/barriers.h`.

```
void IoReadBarrier(void);
```

Функция добавляет барьер чтения из памяти. Linux-аналог: `rmb()`.

IoReadWriteBarrier()

Функция объявлена в файле `coresrv/io/barriers.h`.

```
void IoReadWriteBarrier(void);
```

Функция добавляет обобщенный барьер. Linux-аналог: `mb()`.

IoWriteBarrier()

Функция объявлена в файле `coresrv/io/barriers.h`.

```
void IoWriteBarrier(void);
```

Функция добавляет барьер записи. Linux-аналог: `wmb()`.

Получение сведений об использовании процессорного времени и памяти

Библиотека `libkos` предоставляет API, который позволяет получить сведения об использовании процессорного времени и памяти. Этот API определен в заголовочном файле `sysroot-*-kos/include/coresrv/stat/stat_api.h` из состава KasperskyOS SDK.

> Чтобы получить сведения об использовании процессорного времени и памяти, а также другие 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 процессора отсчитывается с момента запуска ядра KasperskyOS.

Чтобы получить сведения об использовании процессорного времени, нужно использовать функции `KnGroupStatGetParam()` и `KnTaskStatGetParam()`. 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 процессора в режиме ядра
`KnGroupStatGetParam()`	`GROUP_PARAM_CPU_USER`	Время работы процессора в

		пользовательском режиме
`KnGroupStatGetParam()`	`GROUP_PARAM_CPU_IDLE`	Время работы процессора в режиме бездействия
`KnTaskStatGetParam()`	`TASK_PARAM_TIME_TOTAL`	Время работы процессора, затраченное на исполнение процесса
`KnTaskStatGetParam()`	`TASK_PARAM_TIME_USER`	Время работы процессора, затраченное на исполнение процесса в пользовательском режиме

Время работы процессора, полученное вызовом функции `KnGroupStatGetParam()` или `KnTaskStatGetParam()`, выражено в наносекундах.

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 оперативной памяти
`KnGroupStatGetParam()`	`GROUP_PARAM_MEM_FREE`	Размер свободной оперативной памяти
`KnTaskStatGetParam()`	`TASK_PARAM_MEM_PHY`	Размер оперативной памяти, используемой процессом

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 оперативной 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.

Перечисление процессов

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 не использовать 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 процессора

Показателями загрузки процессора могут быть процент общей загрузки процессора и процент загрузки процессора каждым процессом. Расчет этих показателей выполняется для интервала времени, в начале и конце которого были получены сведения об использовании процессорного времени. (Например, может выполняться мониторинг загрузки процессора 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 процессора в режиме ядра;
- *TU* – время работы процессора в пользовательском режиме;
- *TIDLE* – время работы процессора в режиме бездействия;
- *Ti [i=1,2,...,n]* – процессорное время, затраченное на исполнение *i*-го процесса.

Процент общей загрузки процессора рассчитывается так:

(TK+TU)/(TK+TU+TIDLE).

Процент загрузки процессора *i*-м процессом рассчитывается так:

Ti/(TK+TU+TIDLE).

Получение дополнительных сведений о процессах

Помимо сведений об использовании процессорного времени и памяти функции `KnGroupStatGetParam()` и `KnTaskStatGetParam()` позволяют получить, например, такие сведения:

- число процессов;
- число потоков исполнения;
- число потоков исполнения в одном процессе;
- идентификатор родительского процесса (PPID);
- приоритет процесса;

- число дескрипторов, которыми владеет процесс;
- размер виртуальной памяти процесса.

Функция `KnTaskStatGetId()` позволяет получить идентификатор процесса (PID).

Отправка и прием IPC-сообщений

Call()

Функция объявлена в файле `coresrv/syscalls.h`.

```
Retcode Call(Handle handle, const SMsgHdr *msgOut, SMsgHdr *msgIn);
```

Функция отправляет IPC-запрос серверному процессу и блокирует вызывающий поток до получения IPC-ответа или ошибки. Функция вызывается клиентским процессом.

Параметры:

- `handle` – клиентский IPC-дескриптор используемого канала;
- `msgOut` – буфер, содержащий IPC-запрос;
- `msgIn` – буфер под IPC-ответ.

Возвращаемое значение:

- **rcOk** – обмен IPC-сообщениями успешно завершен;
- **rcInvalidArgument** – IPC-запрос и/или IPC-ответ имеют некорректную структуру;
- **rcSecurityDisallow** – отправка IPC-запроса или IPC-ответа запрещена модулем безопасности KSM;
- **rcNotConnected** – серверный IPC-дескриптор канала не найден.

Возможны другие коды возврата.

Recv()

Функция объявлена в файле `coresrv/syscalls.h`.

```
Retcode Recv(Handle handle, SMsgHdr *msgIn);
```

Функция блокирует вызывающий поток до получения IPC-запроса. Функция вызывается серверным процессом.

Параметры:

- `handle` – серверный IPC-дескриптор используемого канала;
- `msgIn` – буфер под IPC-запрос.

Возвращаемое значение:

- **rcOk** – IPC-запрос успешно получен;
- **rcInvalidArgument** – IPC-запрос имеет некорректную структуру;
- **rcSecurityDisallow** – отправка IPC-запроса запрещена модулем безопасности KSM.

Возможны другие коды возврата.

Reply()

Функция объявлена в файле `coresrv/syscalls.h`.

```
Retcode Reply(Handle handle, const SMsgHdr *msgOut);
```

Функция отправляет IPC-ответ и блокирует вызывающий поток до получения ответа клиентом или получения ошибки. Функция вызывается серверным процессом.

Параметры:

- `handle` – серверный IPC-дескриптор используемого канала;
- `msgOut` – буфер, содержащий IPC-ответ.

Возвращаемое значение:

- **rcOk** – IPC-ответ успешно получен клиентом;
- **rcInvalidArgument** – IPC-ответ имеет некорректную структуру;
- **rcSecurityDisallow** – отправка IPC-ответа запрещена модулем безопасности KSM.

Возможны другие коды возврата.

Поддержка POSIX

Ограничения поддержки POSIX

В KasperskyOS ограниченно реализован интерфейс POSIX с ориентацией на стандарт POSIX.1-2008 (без поддержки XSI). Прежде всего ограничения связаны с обеспечением безопасности.

Ограничения затрагивают:

- взаимодействие между процессами;
- взаимодействие между потоками исполнения посредством сигналов;
- стандартный ввод-вывод;
- асинхронный ввод-вывод;
- использование робастных мьютексов;
- работу с терминалом;
- использование оболочки;
- манипуляции с дескрипторами файлов.

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.	Заглушка	`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	Не реализован	`sys/wait.h`

	дочернего процесса.		
`waitpid()`	Ожидать остановки или завершения дочернего процесса.	Заглушка	`sys/wait.h`
`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.	Заглушка	`unistd.h`
`execve()`	Запустить исполняемый файл.	Заглушка	`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.	Не реализован	`unistd.h`
`getpgrp()`	Получить идентификатор группы для вызывающего процесса.	Не реализован	`unistd.h`
`getpgid()`	Получить идентификатор группы.	Заглушка	`unistd.h`
`getppid()`	Получить идентификатор родительского процесса.	Не реализован	`unistd.h`
`getsid()`	Получить идентификатор сессии.	Заглушка	`unistd.h`
`times()`	Получить значения времени	Заглушка	`sys/times.h`

	для процесса и его потомков.		
kill()	Послать сигнал процессу или группе процессов.	Можно посылать только сигнал SIGTERM. Параметр pid игнорируется.	signal.h
pause()	Ожидать сигнала.	Не реализован	unistd.h
sigpending()	Проверить наличие полученных заблокированных сигналов.	Не реализован	signal.h
sigprocmask()	Получить и изменить набор заблокированных сигналов.	Заглушка	signal.h
sigsuspend()	Ожидать сигнала.	Заглушка	signal.h
sigwait()	Ожидать сигнала из заданного набора сигналов.	Заглушка	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 функции ненулевое значение через параметр pshared, то она 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_ mutexattr_ setpshared()	Задать атрибут мьютекса, который разрешает использование мьютекса несколькими процессами.	Нельзя задать атрибут мьютекса, который разрешает использование мьютекса несколькими процессами. Если передать функции значение PTHREAD_PROCESS_SHARED через параметр pshared, то она только вернет значение ENOSYS.	pthread.h
pthread_ barrierattr_ setpshared()	Задать атрибут барьера, который разрешает использование	Нельзя задать атрибут барьера, который разрешает использование барьера несколькими процессами. Если передать функции значение PTHREAD_PROCESS_SHARED	pthread.h

	барьера несколькими процессами.	через параметр `pshared`, то она только вернет значение `ENOSYS`.	
`pthread_condattr_setpshared()`	Задать атрибут условной переменной, который разрешает использование условной переменной несколькими процессами.	Нельзя задать атрибут условной переменной, который разрешает использование условной переменной несколькими процессами. Если передать функции значение `PTHREAD_PROCESS_SHARED` через параметр `pshared`, то она только вернет значение `ENOSYS`.	`pthread.h`
`pthread_rwlockattr_setpshared()`	Задать атрибут объекта блокировки чтения-записи, который разрешает использование объекта блокировки чтения-записи несколькими процессами.	Нельзя задать атрибут объекта блокировки чтения-записи, который разрешает использование объекта блокировки чтения-записи несколькими процессами. Если передать функции значение `PTHREAD_PROCESS_SHARED` через параметр `pshared`, то она только вернет значение `ENOSYS`.	`pthread.h`
`pthread_spin_init()`	Создать спин-блокировку.	Нельзя создать спин-блокировку для синхронизации между процессами. Если передать функции значение `PTHREAD_PROCESS_SHARED` через параметр `pshared`, то это значение будет проигнорировано.	`pthread.h`
`shm_open()`	Создать или открыть объект разделяемой памяти.	Не реализован	`sys/mman.h`
`mmap()`	Отобразить в память.	Нельзя выполнить отображение в память для взаимодействия между процессами. Если передать функции значения `MAP_SHARED` и `PROT_WRITE` через параметры `flags` и `prot` соответственно, то функция вернет значение `MAP_FAILED` и присвоит переменной `errno` значение `EACCES`. Для остальных возможных значений параметра `prot` значение `MAP_SHARED` параметра `flags` будет проигнорировано. Кроме того, через параметр `prot` нельзя передавать сочетания флагов `PROT_WRITE|PROT_EXEC` и `PROT_READ|PROT_WRITE|PROT_EXEC`. В этом случае функция только возвращает значение `MAP_FAILED` и присваивает переменной `errno` значение `ENOMEM`.	`sys/mman.h`
`mprotect()`	Задать права доступа к памяти.	По умолчанию функция работает как заглушка. Чтобы использовать функцию, требуется задать специальные параметры ядра KasperskyOS.	`sys/mman.h`

`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 функции номер сигнала через параметр `sig`, то она только вернет значение `ENOSYS`.	`signal.h`
`pthread_sigmask()`	Получить и изменить набор заблокированных сигналов.	Заглушка	`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 стандартного ввода-вывода

Интерфейс	Назначение	Реализация	Заголовочный файл по стандарту POSIX.1-2008
`dprintf()`	Выполнить форматированный вывод в файл.	Не реализован	`stdio.h`
`fmemopen()`	Использовать память как поток данных (stream).	Не реализован	`stdio.h`
`open_memstream()`	Использовать динамически выделенную память как поток данных (stream).	Не реализован	`stdio.h`

`vdprintf()`	Выполнить форматированный вывод в файл.	Не реализован	`stdio.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 выполнения 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.	Функция только возвращает или передает через параметр `s` пустую	`stdio.h`

		строку.	
`tcsetattr()`	Задать параметры терминала.	Скорость ввода, скорость вывода и другие параметры, специфичные для аппаратных терминалов, игнорируются.	`termios.h`
`tcdrain()`	Ожидать завершения вывода.	Функция 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.	Функция только возвращает значение `-1`.	`termios.h`
`tcsendbreak()`	Разорвать соединение с терминалом на заданное время.	Функция только возвращает значение `-1`.	`termios.h`
`ttyname()`	Получить путь к файлу терминала.	Функция только возвращает нулевой указатель.	`unistd.h`
`ttyname_r()`	Получить путь к файлу терминала.	Функция только возвращает значение ошибки.	`unistd.h`
`tcgetpgrp()`	Получить идентификатор группы процессов, использующих терминал.	Функция только возвращает значение `-1`.	`unistd.h`
`tcsetpgrp()`	Задать идентификатор группы процессов, использующих терминал.	Функция только возвращает значение `-1`.	`unistd.h`
`tcgetsid()`	Получить идентификатор группы процессов для лидера сессии, связанной с терминалом.	Функция 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 канал с этим процессом.	Функция только присваивает переменной `errno` значение `ENOSYS` и возвращает значение `NULL`.	`stdio.h`
`pclose()`	Закрыть канал с дочерним процессом, созданным функцией `popen()`, и ожидать завершения этого дочернего процесса.	Функцию нельзя использовать, так как ее входным параметром является дескриптор потока данных, возвращаемый функцией `popen()`, которая не может вернуть ничего, кроме значения `NULL`.	`stdio.h`
`system()`	Создать дочерний процесс для выполнения	Заглушка	`stdlib.h`

	команды.		
`wordexp()`	Раскрыть строку как в оболочке.	Не реализован	`wordexp.h`
`wordfree()`	Освободить память, выделенную для результатов вызова интерфейса `wordexp()`.	Не реализован	`wordexp.h`

Ограничения манипуляций с 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`

Совместное использование POSIX и других интерфейсов

Использование libkos совместно с Pthreads

В потоке исполнения, созданном с помощью Pthreads, нельзя использовать следующие интерфейсы libkos:

- примитивы синхронизации;
- потоки;
- DMA-буферы;
- порты ввода-вывода;
- ввод-вывод через память (MMIO);
- прерывания.

Следующие интерфейсы libkos можно использовать совместно с Pthreads (и другими интерфейсами POSIX):

- дескрипторы;
- уведомления;

- процессы;
- динамическое создание каналов;
- очереди.

Использование POSIX совместно с libkos threads

Методы POSIX нельзя использовать в потоках исполнения, созданных с помощью libkos threads.

Использование IPC совместно с Pthreads/libkos threads

Методы для IPC можно использовать в любых потоках исполнения, созданных с использованием Pthreads или libkos threads.

Компонент MessageBus

Компонент `MessageBus` реализует шину сообщений, которая обеспечивает прием, распределение и доставку сообщений между приложениями, работающими под KasperskyOS. Шина построена по принципу издатель-подписчик. Использование шины сообщений позволяет избежать создания большого количества IPC-каналов для связывания каждого приложения-подписчика с каждым приложением-издателем.

> Сообщения, передаваемые через шину `MessageBus`, не могут содержать данные. Эти сообщения могут использоваться только для уведомления подписчиков о событиях. См. "Структура сообщения" ниже.

Компонент `MessageBus` представляет собой дополнительный уровень абстракции над KasperskyOS IPC, который позволяет упростить процесс разработки и развития приложений прикладного уровня. `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-соединения прикладной 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;
};
```

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`	Аргумент функции некорректен.
`rcNotConnected`	Нет соединения между клиентской и серверной сторонами взаимодействия. Например, отсутствует серверный IPC-дескриптор.
`rcOutOfMemory`	Недостаточно памяти для выполнения операции.
`rcBufferTooSmall`	Слишком маленький буфер.
`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`	Слишком большой буфер.
`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.

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.

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.

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 образа решения".

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 решения его 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`.

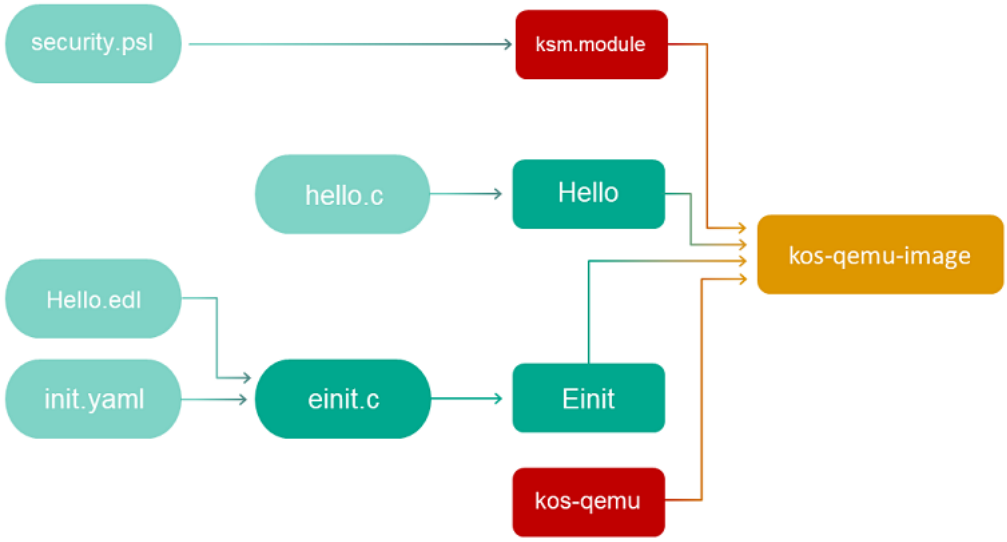
7. Создать образ решения.

   - 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

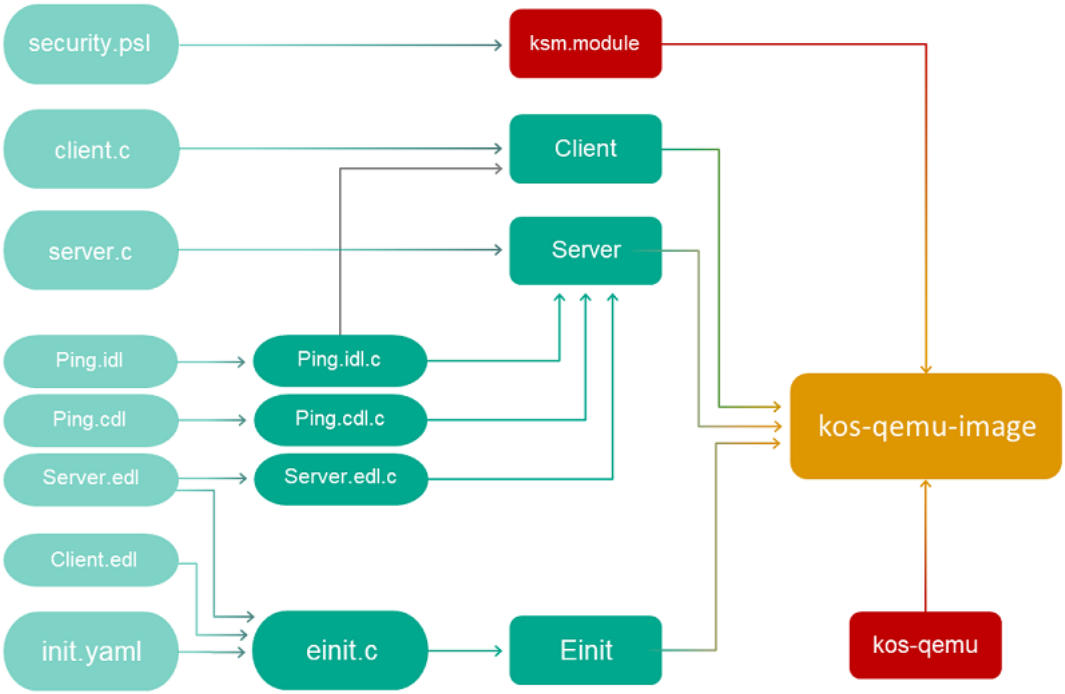
Для простейшего примера `hello`, входящего в состав KasperskyOS Community Edition, в котором содержится одна прикладная программа, не предоставляющая служб, схема сборки выглядит следующим образом:



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, `Ping`), 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 для упрощения использования скриптов `CMake` 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.

- **[Опционально]** Создать скрипт сборки `cross-build.sh`, содержащий команды для запуска генерации файлов сборки (команда `cmake`), сборки решения (команда `make`), а также запуска решения.

Пример структуры директорий проекта

```
example$ tree
.
├── CMakeLists.txt
├── cross-build.sh
├── hello
│   ├── CMakeLists.txt
│   ├── src
│   │   ├── hello.c
├── einit
│   ├── CMakeLists.txt
│   ├── src
│   │   ├── init.yaml.in
│   │   ├── security.psl.in
│   │   ├── fstab
├── resources
│   ├── Hello.idl
│   ├── Hello.cdl
│   ├── Hello.edl
```

Сборка проекта

Для подготовки к сборке с помощью системы сборки `CMake` необходимо:

1. Подготовить корневой файл CMakeLists.txt, содержащий общие инструкции сборки для всего решения.
2. Подготовить файлы CMakeLists.txt для каждой собираемой прикладной программы.
3. Подготовить файл CMakeLists.txt для программы Einit.
4. Подготовить шаблоны `init.yaml.in` и `security.psl.in`.

Для выполнения кросс-компиляции с помощью системы автоматизации сборки `CMake` необходимо:

1. Создать поддиректорию для сборки.

   ```
   BUILD=$PWD/.build
   mkdir -p $BUILD && cd $BUILD
   ```

2. Перед запуском генерации скриптов сборки (команда `cmake`) установить следующие значения переменных окружения:

   - `export LANG=C`
   - `export PKG_CONFIG=""`
   - `export SDK_PREFIX="/opt/KasperskyOS-Community-Edition-<version>"`
   - `export PATH="$SDK_PREFIX/toolchain/bin:$PATH"`

- `export INSTALL_PREFIX=$BUILD/../install`
- `export TARGET="aarch64-kos"`

3. При запуске генерации скриптов сборки (команда `cmake`) указать:
   - параметр `-G "Unix Makefiles"`
   - путь к файлу расширения системы сборки (`toolchain.cmake`) в переменной `CMAKE_TOOLCHAIN_FILE`.

     Файл расширения системы сборки расположен в следующей директории: `/opt/KasperskyOS-Community-Edition-<version>/toolchain/share/toolchain-aarch64-kos.cmake`
   - значение переменной `CMAKE_BUILD_TYPE:STRING=Debug`
   - значение переменной `CMAKE_INSTALL_PREFIX:STRING=$INSTALL_PREFIX`
   - путь к корневому файлу CMakeLists.txt
4. При запуске сборки (команда `make`) указать одну из целей сборки.

   Имя цели должно совпадать с именем цели сборки, переданным в команду сборки решения в файле CMakeLists.txt для программы Einit.

Пример скрипта сборки cross-build.sh

cross-build.sh

```
#!/bin/bash

# Создаем поддиректорию для сборки
BUILD=$PWD/.build
mkdir -p $BUILD && cd $BUILD

# Устанавливаем значения переменных окружения
export LANG=C
export PKG_CONFIG=""
export SDK_PREFIX="/opt/KasperskyOS-Community-Edition-<version>"
export PATH="$SDK_PREFIX/toolchain/bin:$PATH"
export INSTALL_PREFIX=$BUILD/../install
export TARGET="aarch64-kos"

# Запускаем генерацию файлов для сборки. Так как текущая директория это $BUILD,
# корневой файл CMakeLists.txt находится в родительской директории
cmake -G "Unix Makefiles" \
      -D CMAKE_BUILD_TYPE:STRING=Debug \
      -D CMAKE_INSTALL_PREFIX:STRING=$BUILD/../.install \
      -D CMAKE_TOOLCHAIN_FILE=$SDK_PREFIX/toolchain/share/toolchain-$TARGET.cmake \
      ../

# Запускаем сборку. Включаем флаг VERBOSE для make и перенаправляем вывод в файл
build.log
VERBOSE=1 make kos-qemu-image 2>&1 | tee build.log

# Запускаем собранный образ решения в QEMU.
# -kernel $BUILD/einit/kos-qemu-image путь к собранному образу ядра
$SDK_PREFIX/toolchain/bin/qemu-system-aarch64 \
    -m 1024 \
```

```
    -cpu core2duo \
    -serial stdio \
    -kernel $BUILD/einit/kos-qemu-image
```

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 3.12)` – указание минимальной поддерживаемой версии `CMake`.

  Для сборки решения на базе KasperskyOS требуется `CMake` версии не ниже 3.12.

  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.

  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.

  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.

  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.

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, которую далее можно использовать для сборки образа для аппаратной платформы с помощью `make`.
  - build_kos_qemu_image() – создает цель сборки, которую далее можно использовать для сборки образа для запуска на QEMU с помощью `make`.

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
    )

# Образ решения для аппаратной платформы.
# Создает цель сборки с именем kos-image, которую далее можно
# использовать для сборки образа для аппаратной платформы с помощью make kos-image.
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}
                    )

# Образ решения для аппаратной платформы для QEMU.
# Создает цель сборки с именем kos-qemu-image, которую далее можно
# использовать для сборки образа QEMU с помощью make kos-qemu-image.
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:

- описание глобальных параметров политики безопасности решения;
- включение PSL-файлов;
- включение EDL-файлов прикладных программ;
- создание объектов моделей безопасности;
- привязка методов моделей безопасности к событиям безопасности;
- описание 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`.
- `project_header_default()` – команда для указания флагов компилятора и компоновщика для текущего проекта.

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` – название класса процессов. Обязательный параметр.
- `PREFIX` – имя глобального модуля, к которому относится EDL-файл. В этом параметре необходимо указать название проекта.
- `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 `${CMAKE_CURRENT_BINARY_DIR}`.
- `OUTPUT_FILE` – имя создаваемого EDL-файла. По умолчанию `${OUTPUT_DIR}/${NAME}.edl`.

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` – глобальный модуль, к которому относится интерфейс. В этом параметре необходимо указать название проекта.
- `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` – глобальный модуль, к которому относится компонент. В этом параметре необходимо указать название проекта.

- `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` – глобальный модуль, к которому относится EDL-файл. В этом параметре необходимо указать название проекта.
- `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".

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_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, которую впоследствии можно использовать для сборки образа для аппаратной платформы с помощью `make`.

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`.
- `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".

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, которую впоследствии можно использовать для сборки образа для QEMU с помощью `make`.

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`.
- `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".

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 набор транспортных методов и типов на основе EDL-, CDL- и IDL-описаний программ, компонентов и интерфейсов. Транспортные методы и типы нужны 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 основе описания политики безопасности решения (`security.psl`) и EDL, CDL- и IDL-описаний , входящих в решение.

- einit

  Утилита `einit` позволяет автоматизировать создание кода инициализирующего 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 набор транспортных методов и типов на основе EDL-, CDL- и IDL-описаний. Транспортные методы и типы нужны для формирования, отправки, приема и обработки IPC-сообщений.

Компилятор NK принимает EDL-, CDL- или IDL-файл и создает следующие файлы:

- `H`-файл, содержащий объявление и реализацию транспортных методов и типов.
- `D`-файл, в котором перечислены зависимости созданного `C`-файла. Этот файл может быть использован для автоматизации сборки с помощью утилиты `make`.

Синтаксис использования компилятора NK:

```
nk-gen-c [-I PATH][-o PATH][--types][--interface][--client][--server][--extended-errors][--enforce-alignment-check][--help][--version] FILE
```

Параметры:

- `FILE`

  Путь к EDL-, CDL- или IDL-описанию, для которого необходимо сгенерировать транспортные методы и типы.

- `-I PATH`

  Путь к директории, содержащей вспомогательные файлы, необходимые для генерации транспортных методов и типов. По умолчанию эти файлы располагаются в директории `/opt/KasperskyOS-Community-Edition-<version>/sysroot-aarch64-kos/include`.

  Также может использоваться для добавления других директорий для поиска файлов, необходимых для генерации.

  Чтобы указать более одной директории, можно использовать несколько параметров `-I`.

- `-o PATH`

  Путь к существующей директории, в которой будут созданы файлы, содержащие транспортные методы и типы.

- `-h, --help`

  Отображает текст справки.

- `--version`

  Отображает версию `nk-gen-c`

- `--enforce-alignment-check`

Включает обязательную проверку выравнивания обращений к памяти, даже если такая проверка отключена для целевой платформы. Если проверка включена, то компилятор NK добавляет дополнительные проверки выравнивания в код валидаторов IPC-сообщений.

По умолчанию параметры проверки выравнивания обращения к памяти задаются для каждой платформы в файле `system.platform`.

- `--extended-errors`

  Включает расширенную обработку ошибок в коде транспортных методов.

Выборочная генерация

Чтобы уменьшить количество генерируемого компилятором NK кода, можно использовать флаги выборочной генерации. Например, для программ, реализующих службы, удобно использовать флаг `--server`, а для программ, являющихся клиентами служб, удобно использовать флаг `--client`.

> Если ни один из флагов выборочной генерации не указан, компилятор NK создаст все возможные для указанного файла транспортные типы и методы.

Флаги выборочной генерации для IDL-файлов:

- `--types`

  Компилятор создаст только константы и типы, включая переопределенные (`typedef`), из входного IDL-файла, а также типы из импортируемых IDL-файлов, которые используются в типах входного файла.

  При этом константы и переопределенные типы из импортируемых IDL-файлов *не будут* явно включены в генерируемые файлы. Если вам необходимо использовать типы из импортируемых файлов в коде, нужно отдельно сгенерировать H-файлы для каждого такого IDL-файла.

- `--interface`

  Компилятор создаст файлы, создаваемые с флагом `--types`, а также структуры сообщений запросов и ответов для всех методов этой службы.

- `--client`

  Компилятор создаст файлы, создаваемые с флагом `--interface`, а также клиентские прокси-объекты и функции их инициализации для всех методов этой службы.

- `--server`

  Компилятор создаст файлы, создаваемые с флагом `--interface`, а также типы и методы диспетчера этой службы.

Флаги выборочной генерации для CDL-файлов и EDL-файлов:

- `--types`

  Компилятор создаст файлы, создаваемые с флагом `--types` для всех служб, предоставляемых этим компонентом.

  При этом явно включены в генерируемые файлы будут только те типы, которые используются в параметрах интерфейсных методов.

- `--interface`

  Компилятор создаст файлы, создаваемые с флагом `--types` для этого компонента/класса процессов, а также файлы, создаваемые с флагом `--interface` для всех предоставляемых этим компонентом.

- `--client`

  Компилятор создаст файлы, создаваемые с флагом `--interface`, а также клиентские прокси-объекты и функции их инициализации для всех служб, предоставляемых этим компонентом.

- `--server`

  Компилятор создаст файлы, создаваемые с флагом `--interface`, а также типы и методы диспетчера этого компонента/класса процессов и типы и методы диспетчеров для всех служб, предоставляемых этим компонентом.

nk-psl-gen-c

Компилятор `nk-psl-gen-c` генерирует исходный код модуля безопасности Kaspersky Security Module на основе описания политики безопасности решения а также EDL-, CDL и IDL-описаний, входящих в решение. Этот код используется скриптом makekss.

Компилятор `nk-psl-gen-c` также позволяет генерировать и запускать код тестов для политики безопасности решения, написанных на языке PAL.

Синтаксис использования компилятора `nk-psl-gen-c`:

```
nk-psl-gen-c [-I PATH][-o PATH][--audit PATH][--tests ARG][--help][--version] FILE
```

Параметры:

- `FILE`

  Путь к PSL-описанию политики безопасности решения (`security.psl`)

- `-I,--include-dir PATH`

  Путь к директории, содержащей вспомогательные файлы, необходимые для генерации транспортных методов и типов. По умолчанию эти файлы располагаются в директории `/opt/KasperskyOS-Community-Edition-<version>/sysroot-aarch64-kos/include`.

  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 `-I`.

  Чтобы указать более одной директории, можно использовать несколько параметров `-I`.

- `-o,--output PATH`

  Путь к создаваемому файлу, содержащему код модуля безопасности.

- `-t, --tests ARG`

  Флаг контроля генерации кода и запуска тестов для политики безопасности решения. Может принимать следующие значения:

  - `skip` – код тестов не генерируется. Это значение используется по умолчанию, если флаг `--tests` не указан.
  - `generate` – код тестов генерируется, но не компилируется и не запускается.
  - `run` – код тестов генерируется, компилируется с помощью компилятора `gcc` и запускается.

- `-a, --audit PATH`

  Путь к создаваемому файлу, содержащему код декодера аудита.

- `-h, --help`

  Отображает текст справки.

- `--version`

  Отображает версию `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

Свойства кросс-компиляторов KasperskyOS

Кросс-компиляторы, входящие в состав KasperskyOS Community Edition, поддерживают процессоры с архитектурой `aarch64`.

В toolchain в составе KasperskyOS Community Edition входят следующие инструменты для кросс-компиляции:

- GCC:
  - `aarch64-kos-gcc`
  - `aarch64-kos-g++`
- Binutils:
  - Ассемблер AS: `aarch64-kos-as`
  - Компоновщик LD: `aarch64-kos-ld`

В GCC, кроме стандартных макросов, определен дополнительный макрос `__KOS__=1`. Использование этого макроса упрощает портирование программного кода на KasperskyOS, а также разработку платформонезависимых приложений.

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
```

Создание загрузочного носителя с образом решения

Чтобы создать загрузочный носитель с образом решения:

1. Подключите носитель, с которого вы планируете запускать образ решения на целевых устройствах.
2. Найдите блочное устройство, соответствующее подключенному носителю, например с помощью команды:

   `fdisk -l`
3. При необходимости создайте на носителе новый раздел, в котором будет развернут образ решения, например с помощью утилиты `fdisk`.
4. Если в разделе отсутствует файловая система, создайте ее, например с помощью утилиты `mkfs`.

   Вы можете использовать любую файловую систему, которую поддерживает загрузчик GRUB 2.
5. Смонтируйте носитель.

   `mkdir /mnt/kos_device && mount /dev/sdXY /mnt/kos_device`

   Здесь `/mnt/kos_device` – точка монтирования; `/dev/sdXY` – имя блочного устройства; `X` – буква, соответствующая подключенному носителю; `Y` – номер раздела.
6. Установите на носитель загрузчик операционной системы GRUB 2.

   Чтобы установить GRUB 2, выполните следующую команду:

   ```
   grub-install --force --removable \
   --boot-directory=/mnt/kos_device/boot /dev/sdX
   ```

   Здесь `/mnt/kos_device` – точка монтирования `/dev/sdX` – имя блочного устройства; `X` – буква, соответствующая подключенному носителю.
7. Скопируйте образ решения в корневую директорию смонтированного носителя.
8. В файле `/mnt/kos_device/boot/grub/grub.cfg` добавьте секцию `menuentry`, указывающую на образ решения.

   ```
   menuentry "KasperskyOS" {
   multiboot /my_kasperskyos.img
   boot
   }
   ```
9. Размонтируйте носитель.

   `umount /mnt/kos_device`

   Здесь `/mnt/kos_device` – точка монтирования.

После выполнения этих действий вы можете запускать KasperskyOS с этого носителя.

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 компоненты, для которых процесс заданного класса является контейнером.

Каждый компонент из состава компонента решения соответствует CDL-описанию. Это описание задает имя компонента, предоставляемые службы, интерфейс безопасности, а также вложенные компоненты. Компоненты могут одновременно предоставлять службы, поддерживать интерфейс безопасности и являться контейнерами для других компонентов. Каждый компонент может предоставлять несколько служб с одним или несколькими интерфейсами.

Каждый интерфейс 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 содержат компонентов, могут быть только клиентами. Процессы, содержащие компоненты, могут быть серверами и/или клиентами. Если компоненты из состава процесса предоставляют службы, то процесс может быть сервером, но и одновременно может быть клиентом. Если компоненты из состава процесса не предоставляют службы (только поддерживают интерфейс безопасности), то процесс может быть только клиентом.

> 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. Имена классов процессов, компонентов и пакетов образуют в рамках решения на базе KasperskyOS три множества имен, в которых любое имя является уникальным в рамках своего множества. 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 файлам>`.) 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`, которые содержат следующие данные:

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.

Язык EDL чувствителен к регистру символов.

В EDL-описании могут использоваться однострочные и многострочные комментарии.

> В EDL-описании могут задаваться поддерживаемые службы и интерфейс безопасности точно так же, как они задаются в CDL-описании. Такая практика не рекомендуется, так как в общем случае EDL- и CDL-описания создаются разными участниками процесса разработки решения на базе KasperskyOS. CDL-описания создаются разработчиками системного и прикладного ПО. EDL-описания создаются разработчиками, которые выполняют интеграцию системного и прикладного ПО в единое решение.

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
}
```

CDL-описание

CDL-описания помещаются в отдельные файлы `*.cdl`, которые содержат следующие данные:

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`, которые содержат следующие данные:

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`).

   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

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);
- `string<<размер в байтах>>` (строковый буфер).

Байтовый буфер представляет собой область памяти с размером, не превышающим заданного числа байт. Строковый буфер представляет 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. Для передачи байтового или строкового буфера через IPC будет задействовано столько памяти, сколько фактически занимает этот буфер.

Для числовых типов можно объявлять именованные константы с помощью ключевого слова `const`:

```
const UInt32 DeviceNameMax          = 64;
const UInt32 HandleTypeUserLast     = 0x0001FFFF;
```

Константы используются, чтобы избежать проблемы "магических чисел". К примеру, если в IDL-описании определены константы для кодов возврата интерфейсного метода, то при описании политики можно интерпретировать эти коды без дополнительных сведений.

Помимо примитивных типов в языке IDL поддерживаются составные типы: объединения, структуры, массивы и последовательности. В определении составных типов константы примитивных типов могут применяться как параметры (например, чтобы задать размер массива).

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.

Последовательности

Последовательность представляет собой массив переменного размера. При определении последовательности указывается максимальное число ее элементов, однако фактически можно передать (через IPC) меньшее их число. При этом для передачи будет задействовано столько памяти, сколько занимают передаваемые элементы.

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.

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:

```
struct BazInfo {
    array<UInt8, 100> a;
    sequence<sequence<UInt32, 100>, 200> 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 может быть включено заранее описанное определение объединения или структуры. Это выполняется посредством указания в определении типа имен включаемых типов:

```
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:

```
    Poll(in Generation generation,
         in UInt32 timeout,
         out Generation currentGeneration,
         out sequence<Report, DeviceMax> report,
         out UInt32 count,
         out UInt32 rc);
```

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. 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 этот объект будет независимо использовать контексты безопасности этих ресурсов. 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).

Аудит 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 выводит полученные данные через стандартный вывод или сохраняет 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.

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. Вторая и последующие строки декларации должны быть записаны с отступами относительно первой строки. Закрывающая фигурная скобка, которая завершает декларацию, может быть записана на уровне первой строки.

3. В многострочной декларации используются отступы разных размеров, чтобы отразить вложенность конструкций, составляющих эту декларацию. Вторая и последующие строки многострочной конструкции, заключенные в фигурные скобки, должны быть записаны с отступом относительно первой строки этой конструкции. Закрывающая фигурная скобка многострочной конструкции может быть записана с отступом или на уровне первой строки конструкции.

4. Язык PSL чувствителен к регистру символов.

5. Поддерживаются однострочные и многострочные комментарии:

```
/* Это комментарий
 * И это тоже */
// Ещё один комментарий
```

Типы деклараций

В языке PSL есть следующие типы деклараций:

- описание глобальных параметров политики безопасности решения;
- включение PSL-файлов;
- включение EDL-файлов;
- создание объектов моделей безопасности;
- привязка методов моделей безопасности к событиям безопасности;
- описание профилей аудита безопасности;
- описание тестов политики безопасности решения.

Описание 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 другими процессами. Чтобы задать этот интерфейс, нужно использовать 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. Чтобы задать эти параметры, нужно использовать 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.

Включение PSL-файлов

Для включения PSL-файла нужно использовать 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 файлам>` 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 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

Чтобы включить EDL-файл для ядра KasperskyOS, нужно использовать декларацию:

```
use EDL kl.core.Core
```

Чтобы включить EDL-файл для программы (например, для драйвера или прикладной программы), нужно использовать 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 файлам>` 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-файлы, а CDL-файлы содержат ссылки на соответствующие CDL-, IDL-файлы.

Создание объектов моделей безопасности

Для вызова методов требуемой модели безопасности, нужно создать объект этой модели безопасности.

Чтобы создать объект модели безопасности, нужно использовать декларацию:

```
policy object <имя объекта модели безопасности : название модели безопасности> {
    [параметры объекта модели безопасности]
}
```

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 декларацию:

```
<вид события безопасности> [селекторы события безопасности] {
    [профиль аудита безопасности]
    <вызываемые правила моделей безопасности>
}
```

Вид события безопасности

Чтобы задать вид события безопасности, используются 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. Используются 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 составляют 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

Профиль аудита безопасности задается конструкцией `audit <имя профиля аудита безопасности>`. Если профиль аудита безопасности не задан, используется глобальный профиль аудита безопасности.

Вызываемые правила моделей безопасности

Вызываемые правила моделей безопасности задаются списком из конструкций следующего вида:

```
[имя объекта модели безопасности.]<имя правила модели безопасности> <параметр>
```

Входными данными для правил моделей безопасности могут быть значения, возвращаемые выражениями моделей безопасности. Для вызова выражения модели безопасности используется конструкция:

```
[имя объекта модели безопасности.]<имя выражения модели безопасности> <параметр>
```

Также в качестве входных данных для методов моделей безопасности (правил и выражений) могут использоваться параметры интерфейсных методов. (О получении доступа к параметрам интерфейсных методов см. "Модель безопасности Struct"). Кроме этого, входными данными для методов моделей безопасности могут быть 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 вызовов некоторых правил и выражений моделей безопасности можно не указывать имя объекта модели безопасности, а также можно использовать операторы. 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 вида:

```
match <селекторы события безопасности> {
    [профиль аудита безопасности]
    <вызываемые правила 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 конструкции:

```
choice <вызов выражения модели безопасности, проверяющего выполнение условий> {
    "<условие 1>" : [{] // Условная секция 1
        [профиль аудита безопасности]
        <вызываемые правила 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, но можно задать отдельный профиль аудита безопасности для условной секции.

Если при обработке события безопасности выполняется сразу несколько условий, описанных в конструкции `choice`, то срабатывает только одна условная секция, соответствующая первому в списке подходящему 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").

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".

Описание профилей аудита 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. "Описание 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 или choice-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 описать профиль аудита безопасности, нужно использовать декларацию:

```
audit profile <имя профиля аудита безопасности> =
  { <уровень аудита безопасности> :
      // Описание конфигурации аудита безопасности
      { <имя объекта модели безопасности> :
          { kss : <условия выполнения аудита безопасности, связанные с результатами
                   вызовов правил 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 используется 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. "Описание 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).

Уровни аудита можно использовать, например, чтобы регулировать детализацию аудита. Чем выше уровень аудита, тем выше детализация. Чем выше детализация, тем больше объектов моделей безопасности покрывается аудитом и/или меньше ограничений применяется в условиях выполнения аудита.

Другим примером применения уровней аудита является возможность переключать аудит с одной подсистемы на другую (например, переключить аудит, связанный с драйверами, на аудит, связанный с прикладными программами, или аудит, связанный с сетевой подсистемой, на аудит, связанный с графической подсистемой).

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 задаются отдельно для каждого объекта модели безопасности.

Чтобы задать условия выполнения аудита, связанные с результатами вызовов правил моделей безопасности, нужно использовать следующие конструкции:

- `["granted"]` – аудит выполняется, если правила возвращают результат "разрешено";
- `["denied"]` – аудит выполняется, если правила возвращают результат "запрещено";
- `["granted", "denied"]` – аудит выполняется, если правила возвращают результат "разрешено" или "запрещено";
- `[]` – аудит не выполняется независимо от того, какой результат возвращают правила.

Условия выполнения аудита, связанные с результатами вызовов правил, не применяются к выражениям. Эти условия должны быть заданы (любой возможной конструкцией), даже если модель безопасности содержит только выражения, поскольку этого требует синтаксис языка PSL.

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 события безопасности), нужно назначить пустой профиль аудита `empty` на уровне привязки методов моделей безопасности к событиям безопасности или на уровне match-секции либо choice-секции.

Примеры описаний профилей аудита

См. "Примеры описаний профилей аудита безопасности".

Описание 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 описать 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 описать несколько наборов тестов, используя несколько таких деклараций.

Описание набора 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 можно проверить, 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.

Описание тестового примера создается на языке PAL и представляет собой последовательность значений:

```
[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"). Если ожидаемое решение модуля безопасности не указано, 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).

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 не обеспечивать это.)

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> ! <квалифицированно 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>`.

Если у интерфейсного метода есть параметры, то их значения задаются разделенными запятой конструкциями:

```
<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.

Примеры описаний тестов политик

См. "Примеры описаний тестов политик безопасности решений на базе KasperskyOS".

Тестовая процедура

Описания тестов помещаются в PSL-файлы, включая те, которые содержат описание политики безопасности решения (например, в файл `security.psl`).

Чтобы выполнить тесты, нужно использовать параметр `--tests run` при запуске компилятора `nk-psl-gen-c`:

```
$ nk-psl-gen-c --tests run <остальные параметры> security.psl
```

Также компилятору `nk-psl-gen-c` нужно указать следующие сведения:

- Директории, которые содержат вспомогательные файлы из состава KasperskyOS SDK (`common`, `sysroot-*-kos/include`, `toolchain/include`). Этот набор директорий задается параметрами `-I, -include-dir <путь к файлам>`.
- Директории, которые содержат PSL-, IDL-, CDL-, EDL-файлы, относящиеся к решению. Этот набор директорий задается параметрами `-I, --include-dir <путь к файлам>`.
- Путь к файлу, в который будет сохранен исходный код модуля безопасности Kaspersky Security Module и тестов. Этот путь задается параметром `-o, --output <путь к файлу>`.

Компилятор `nk-psl-gen-c` генерирует исходный код модуля безопасности и тестов на языке C, сохраняет его в файл, а затем запускает компиляцию этого кода с использованием gcc и выполнение полученной тестовой программы. Тестовая программа запускается в среде, где установлен KasperskyOS SDK, то есть на компьютере под управлением ОС Linux. Ядро KasperskyOS, а также системное и прикладное ПО решения не используются.

Чтобы сгенерировать исходный код модуля безопасности и тестов, но не компилировать его, нужно использовать параметр `--tests generate` при запуске компилятора `nk-psl-gen-c`.

По умолчанию результаты тестирования выводятся в консоль. Чтобы вывести результаты тестирования в файл, нужно использовать параметр `--test-output <путь к файлу>` при запуске компилятора `nk-psl-gen-c`.

Пример результатов тестирования:

```
# PAL test run

## Execute (1/2)

* Happy path: FAIL
  Step 2/2: ExpectGrant Execute "This should not fail"
  component/secure_platform/kss/nk/psl/nk-psl-gen-
c/tests/examples/include/router.psl:38:5-40:3
* No rule: PASS

## IPC (2/2)

* Happy path: PASS
* No rule: PASS

## Security (2/2)

* Happy path: PASS
* No rule: PASS
```

Результаты тестирования содержат сведения о том, прошел или не прошел каждый тест. Если тест не прошел, то указывается, какой тестовый пример из этого теста не прошел, а также 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 () }
    }
}
```

Задание профилей аудита

```
/* Задание глобального профиля аудита default
 * и начального уровня аудита 0 */
audit default = global 0
request src=Client, dst=Server {
    /* Задание профиля аудита parent на уровне
     * привязки методов моделей безопасности к
     * событиям безопасности */
    audit parent
    match endpoint=net.Net, method=Send {
        /* Задание 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 описаний 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 результат "разрешено"
 * или "запрещено", и конечный автомат находится в состоянии,
 * отличном от closed.
 * 2. Выражение query объекта session вызывается, 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 результат "разрешено"
 * или "запрещено".
 * 2. Выражение query объекта session вызывается. */
  , 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 : []
        , 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 : []
        , emit : ["match", "select"]
        }
      }
  }
```

Примеры описаний тестов политик безопасности решений на базе KasperskyOS

Пример 1

```
/* Описание набора тестов, который включает один тест. */
assert "some tests" {
    /* Описание теста, который включает четыре тестовых примера. */
    sequence "first sequence" {
        /* Ожидается, что запуск процесса класс Server разрешен.
         * Если это так, переменной s будет присвоено значение SID
         * запущенного процесса класса Server. */
        s <- execute dst=Server
        /* Ожидается, что запуск процесса класс 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 cannot respond" response src=s dst=c endpoint=pingComp.pingImpl
                                                 method=Ping {}
    }
}
```

Пример 2

```
/* Описание набора тестов, который включает два теста. */
assert "ping tests"{
    /* Начальная часть каждого из двух тестов */
    setup {
        s <- execute dst=Server
        c <- execute dst=Client
    }
/* Описание теста, который включает два тестовых примера. */
    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 }
    }
/* Описание теста, который включает два тестовых примера. */
    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

```
/* Описание набора тестов, который включает один тест. */
```

```
assert {
    /* Описание теста, 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 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 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 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 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 конструкцию:

```
bool.any <List<Boolean>>
```

Выражение `cond` выполняет тернарную условную операцию. Возвращает значения типа `ScalarLiteral`. Чтобы вызвать выражение, нужно использовать 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 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 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:

- `<{...}>.<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 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 `Kaspersky0S`.

- `<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 начальный 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 является 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_uncommited модели безопасности StaticMap

```
get_uncommited {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"]
                             }
             }
}
```

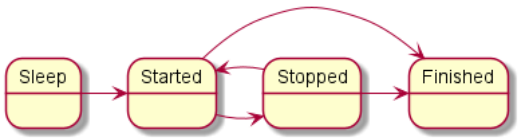


Диаграмма состояний конечного автомата в примере

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 назначать субъектам и ресурсам уровни целостности, 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 значение `container` не задано (`container : ()`), 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 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 значение `level` не задано (`level : ()`), субъекту `target` назначается уровень целостности исполняемого файла `image`. Если значение `image` не задано (`image : ()`), должно быть задано значение `level`.

Если значение `levelR` не задано (`levelR : ()`), то значение `levelR` равно `level`.

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 значение `container` не задано (`container : ()`), 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
	Назначение	Позволяет выполнить следующие

`Allocate`	Выделяет (резервирует и опционально отображает на физическую память) 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 и его выделения. • [out] `va` – базовый адрес выделенного региона виртуальной памяти. • [out] `rc` – код возврата.	действия: • Исчерпать память ядра, создавая в ней множество объектов. • Исчерпать оперативную память.
`Commit`	Назначение Отображает зарезервированный методом `Allocate` регион виртуальной памяти (или его часть) на физическую память. Параметры • [in] `va` – базовый адрес региона виртуальной памяти. • [in] `size` – размер региона виртуальной памяти в байтах. • [in] `flags` – флаги, задающие параметры региона виртуальной памяти. • [out] `rc` – код возврата.	Позволяет исчерпать оперативную память.
`Decommit`	Назначение Отменяет отображение региона виртуальной памяти на физическую память. Параметры • [in] `va` – базовый адрес региона виртуальной памяти.	Нет.

	• [in] `size` – размер региона виртуальной памяти в байтах. • [out] `rc` – код возврата.	
`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 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.	Нет.

`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 буфера MDL в байтах. • [in] `flags` – 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.
`MdlGetSize`	Назначение Позволяет получить размер буфера MDL.	Нет.

	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.	
`MdlMap`	Назначение Отображает буфер MDL на регион виртуальной памяти. 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. • [in] `prot` – флаги, задающие права доступа к региону виртуальной памяти. • [out] `address` – базовый адрес 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 MDL владеют несколько процессов (маски прав дескрипторов должны разрешать отображение буфера MDL). • Исчерпать память ядра, создавая в ней множество объектов.
`MdlClone`	Назначение Создает буфер MDL на основе существующего.	Позволяет исчерпать память ядра, создавая в ней множество объектов.

	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. • [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.	

Служба ввода-вывода

Служба предназначена для работы с портами ввода-вывода, MMIO, DMA, прерываниями.

Сведения о методах службы приведены в таблице ниже.

Методы службы io.IO (интерфейс kl.core.IO)

Метод	Назначение и параметры метода	Потенциальная опасность метода
`RegisterPort`	Назначение Регистрирует порты ввода-вывода. Параметры • [in] `base` – базовый адрес портов ввода-вывода. • [in] `size` – ширина диапазона адресов портов ввода-вывода.	Позволяет выполнить следующие действия: • Захватить порты ввода-вывода (рекомендуется контролировать базовый адрес и ширину диапазона адресов для портов ввода-вывода). • Исчерпать память ядра, создавая в ней множество объектов.

	• [out] `resource` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует порты ввода-вывода. • [out] `rc` – код возврата.	
`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 DMA. • [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.	Позволяет выполнить следующие действия: • Исчерпать память ядра, создавая в ней множество объектов. • Исчерпать оперативную память.

`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 Отображает регион памяти MMIO на регион виртуальной памяти. 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 идентифицирует регион виртуальной памяти. • [out] `rc` – код возврата.	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 множество объектов.
`PermitPort`	Назначение Открывает доступ к портам ввода-вывода. Параметры	Позволяет выполнить следующие действия: • Захватить управление устройством (рекомендуется контролировать базовый адрес и

	• [in] `resource` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует порты ввода-вывода. • [out] `access` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор используется для доступа к портам ввода-вывода. • [out] `rc` – код возврата.	ширину диапазона адресов для портов ввода-вывода при вызове метода `RegisterPort`). • Исчерпать память ядра, создавая в ней множество объектов.
`AttachIrq`	Назначение Привязывает прерывание к дескриптору, используемому обработчиком прерывания. 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, отражающие характеристики прерывания. • [out] `delivery` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор 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). • Создать вредоносную обработку прерывания, например, некорректную обработку или задержку обработки (рекомендуется контролировать номер прерывания при вызове метода `RegisterIrq`). • Выполнить привязку к прерыванию, которое уже привязано к обработчику прерывания в другом процессе, чтобы заблокировать обработку этого прерывания.

		• Исчерпать память ядра, создавая в ней множество объектов.
`AttachIrqEx`	Назначение Привязывает прерывание к дескриптору, используемому обработчиком прерывания. 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, отражающие характеристики прерывания. • [in] `futexPtr` – указатель на фьютекс. • [out] `delivery` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор 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). • Создать вредоносную обработку прерывания, например, некорректную обработку или задержку обработки (рекомендуется контролировать номер прерывания при вызове метода `RegisterIrq`). • Выполнить привязку к прерыванию, которое уже привязано к обработчику прерывания в другом процессе, чтобы заблокировать обработку этого прерывания. • Исчерпать память ядра, создавая в ней множество объектов.
`DetachIrq`	Назначение Отвязывает прерывание от дескриптора, используемого обработчиком прерывания. Параметры • [in] `resource` – значение, двоичное представление которого состоит из нескольких полей, включая поле	Нет.

	дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует прерывание. • [out] `rc` – код возврата.	
`EnableIrq`	Назначение Возобновляет обработку 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.	Нет.
`DisableIrq`	Назначение Блокирует обработку 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 заблокировать обработку прерывания в другом процессе.
`ModifyDma`	Назначение Изменяет параметры 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] `flags` – флаги, задающие параметры DMA. • [out] `rc` – код возврата.	Нет.
`MapDma`	Назначение Отображает буфер DMA на регион виртуальной памяти.	Позволяет выполнить следующие действия: • Создать разделяемую память для межпроцессного

	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 идентифицирует регион виртуальной памяти. • [out] `rc` – код возврата.	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 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.	Нет.

	• [out] `size` – размер буфера DMA в байтах. • [out] `count` – число блоков. • [out] `frames` – последовательность, содержащая сведения о блоках. • [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 непрерывных регионов физической памяти. • [out] `frames` – последовательность, содержащая сведения о непрерывных регионах физической памяти. • [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 прав дескриптора. Дескриптор идентифицирует объект ядра, который используется для отображения буфера	Позволяет исчерпать память ядра, создавая в ней множество объектов.

	DMA на диапазон IOMMU-адресов, используемых устройством. • [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] `tid` – идентификатор потока исполнения (TID). • [in] `priority` – значение, задающее приоритет потока исполнения. • [in] `stackSize` – размер стека для потока исполнения или `0`, чтобы был задан размер по умолчанию. • [in] `routine` – указатель на функцию, которая будет выполнена при создании потока исполнения. • [in] `context` – указатель на функцию, которая будет выполнена в контексте потока 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, которые будут переданы функции, заданной через параметр `context`. • [in] `flags` – флаги, задающие параметры создания потока исполнения. • [out] `rc` – код возврата.	
`Suspend`	Назначение Блокирует поток исполнения. Параметры • [in] `tid` – идентификатор потока исполнения (TID). • [out] `rc` – код возврата.	Позволяет заблокировать стандартный поток исполнения, который захватил объект синхронизации, ожидаемый потоком исполнения реального времени, в контексте которого обрабатывается прерывание. Это может привести к остановке обработки этого прерывания другими процессами.
`Resume`	Назначение Возобновляет поток исполнения. Параметры • [in] `tid` – идентификатор потока исполнения (TID). • [out] `rc` – код возврата.	Нет.
`Terminate`	Назначение Завершает поток исполнения. Параметры	Нет.

	• [in] `tid` – идентификатор потока исполнения (TID). • [in] `zombie` – фиктивный параметр. • [in] `code` – код завершения потока исполнения. • [out] `rc` – код возврата.	
`Exit`	Назначение Завершает текущий поток исполнения. Параметры • [in] `zombie` – фиктивный параметр. • [in] `code` – код завершения потока исполнения. • [out] `rc` – код возврата.	Нет.
`Wait`	Назначение Блокирует текущий поток исполнения до завершения заданного потока исполнения. Параметры • [in] `tid` – идентификатор потока исполнения (TID). • [in] `msec` – время ожидания завершения потока исполнения в миллисекундах.	Нет.

	• [out] `code` – код завершения потока исполнения. • [out] `rc` – код возврата.	
`SetPriority`	Назначение Задает приоритет потока исполнения. Параметры • [in] `tid` – идентификатор потока исполнения (TID). • [in] `priority` – значение, задающее 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.
`GetTcb`	Назначение Позволяет получить доступ к локальной памяти текущего потока исполнения (TLS текущего потока исполнения). Параметры • [out] `va` – указатель на локальную память текущего потока исполнения. • [out] `rc` – код возврата.	Нет.
`SetTls`	Назначение Задает базовый адрес локальной памяти текущего потока исполнения (TLS текущего потока исполнения).	Нет.

	Параметры • [in] `va` – указатель на локальную память текущего потока исполнения. • [out] `rc` – код возврата.	
`Sleep`	Назначение Блокирует текущий поток исполнения на заданное время. Параметры • [in] `mdelay` – время блокировки текущего потока исполнения в миллисекундах. • [out] `rc` – код возврата.	Нет.
`GetInfo`	Назначение Позволяет получить сведения о потоке исполнения. Параметры • [in] `tid` – идентификатор потока исполнения (TID). • [out] `info` – структура со сведениями о потоке исполнения. • [out] `rc` – код возврата.	Нет.
`DetachIrq`	Назначение Отвязывает текущий поток исполнения от прерывания, обрабатываемого в его контексте.	Нет.

	Параметры • [out] `rc` – код возврата.	
`GetAffinity`	Назначение Позволяет получить маску сходства потока исполнения. Параметры • [in] `tid` – идентификатор потока исполнения (TID). • [out] `mask` – маска сходства потока исполнения. • [out] `rc` – код возврата.	Нет.
`SetAffinity`	Назначение Задает маску сходства потока исполнения. Параметры • [in] `tid` – идентификатор потока исполнения (TID). • [in] `mask` – маска сходства потока исполнения. • [out] `rc` – код возврата.	Нет.
`SetSchedPolicy`	Назначение Задает класс планирования потока исполнения. Параметры • [in] `tid` – идентификатор	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

	потока исполнения (TID). • [in] `policy` – значение, задающее класс планирования потока исполнения. • [in] `priority` – значение, задающее приоритет потока исполнения. • [in] `param` – объединение, содержащее параметры класса планирования потока исполнения. • [out] `rc` – код возврата.	потокам исполнения, в том числе из других процессов (рекомендуется контролировать приоритет потока исполнения).
`GetSchedPolicy`	Назначение Позволяет получить сведения о классе планирования потока исполнения. Параметры • [in] `tid` – идентификатор потока исполнения (TID). • [out] `policy` – значение, отражающее класс планирования потока исполнения. • [out] `priority` – значение, отражающее приоритет потока исполнения. • [out] `param` – объединение, содержащее параметры класса планирования потока исполнения.	Нет.

	• [out] `rc` – код возврата.	

Служба дескрипторов

Служба предназначена для работы 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 Создает дескриптор на основе существующего. Параметры • [in] `inHandle` – значение, двоичное представление которого состоит из нескольких полей, включая поле оригинального дескриптора и поле маски прав оригинального дескриптора. • [in] `newRightsMask` – маска прав создаваемого дескриптора. • [in] `copyBadge` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует объект контекста передачи ресурса. • [out] `outHandle` – 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.
`CreateUserObject`	Назначение Создает дескриптор пользовательского ресурса. Параметры • [in] `type` – тип дескриптора. • [in] `rights` – маска прав создаваемого дескриптора. • [in] `context` – указатель на контекст пользовательского ресурса.	Позволяет исчерпать память ядра, создавая в ней множество объектов.

	• [in] `ipcChannel` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор является серверным IPC-дескриптором IPC-канала, ассоциированного с пользовательским ресурсом. • [out] `riid` – идентификатор службы (RIID), ассоциированной с пользовательским ресурсом. • [in] `handle` – значение, двоичное представление которого состоит из нескольких полей, включая поле созданного дескриптора и поле маски прав созданного дескриптора. Дескриптор идентифицирует пользовательский ресурс. • [out] `rc` – код возврата.	
`Close`	Назначение Удаляет дескриптор. Параметры • [in] `handle` – значение, двоичное представление которого состоит из нескольких полей, включая поле удаляемого дескриптора и поле маски прав удаляемого дескриптора. • [out] `rc` – код возврата.	Нет.
`Connect`	Назначение Создает и связывает между собой клиентский, серверный и слушающий IPC-дескрипторы. Параметры • [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 IPC-дескриптор, который уже был создан предыдущим вызовом метода, или значение `0xFFFFFFFF` для создания слушающего IPC-дескриптора. • [in] `client` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует клиентский процесс. • [out] `outSrListerner` – созданный слушающий IPC-дескриптор.	Позволяет исчерпать память ядра, создавая в ней множество объектов.

	• [out] `outSrEndpoint` – серверный IPC-дескриптор. • [out] `outClEndpoint` – клиентский IPC-дескриптор. • [out] `rc` – код возврата.	
`Disconnect`	Назначение Разрывает связь между клиентским и серверным IPC-дескрипторами. Параметры • [in] `client` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор является клиентским IPC-дескриптором. • [out] `rc` – код возврата.	Нет.
`SecurityConnect`	Назначение Создает дескриптор и связывает его с интерфейсом безопасности. Параметры • [out] `client` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор используется для обращения к модулю безопасности через интерфейс безопасности. • [out] `rc` – код возврата.	Позволяет исчерпать множество возможных значений дескрипторов процесса ядра.
`SecurityDisconnect`	Назначение Разрывает связь дескриптора с интерфейсом безопасности. Параметры • [in] `client` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор используется для обращения к модулю безопасности через интерфейс безопасности. • [out] `rc` – код возврата.	Нет.
`UidAlloc`	Назначение Выделяет значение уникального идентификатора.	Позволяет исчерпать множество возможных

	Метод используется для обратной совместимости, так как в настоящее время вместо уникальных идентификаторов используются дескрипторы. Параметры • [out] `uid` – значение уникального идентификатора. • [out] `rc` – код возврата.	значений уникальных идентификаторов.
`UidFree`	Назначение Освобождает значение уникального идентификатора. (Это значение требуется освободить, чтобы оно стало доступным для повторного использования.) Метод используется для обратной совместимости, так как в настоящее время вместо уникальных идентификаторов используются дескрипторы. Параметры • [in] `uid` – значение уникального идентификатора. • [out] `rc` – код возврата.	Позволяет освободить значение уникального идентификатора, используемого другим процессом.
`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). • [out] `rc` – код возврата.	Нет.
`Revoke`	Назначение Удаляет 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	Нет.

	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 для отзыва. Корневым узлом этого поддерева является дескриптор, который порожден передачей дескриптора, заданного через параметр `handle`, в ассоциации с объектом контекста передачи ресурса. • [out] `rc` – код возврата.	
`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. • [in] `badgeContext` – указатель на контекст передачи ресурса. • [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.	Позволяет исчерпать память ядра, создавая в ней множество объектов.

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 процесса в байтах. • [in] `priority` – значение, задающее 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. Дескриптор идентифицирует созданный процесс. • [out] `rc` – код возврата.	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.
`LoadSeg`	Назначение Загружает сегмент образа программы в память процесса из буфера MDL. Параметры • [in] `task` – значение, двоичное представление которого состоит из нескольких полей, включая	Позволяет загрузить в память процесса код для последующего исполнения.

	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 образа программы. • [in] `segAttr` – структура, содержащая параметры загрузки сегмента образа программы. • [out] `rc` – код возврата. • [out] `retaddr` – базовый адрес региона виртуальной памяти процесса, куда загружен сегмент образа программы.	
`SetEntry`	Назначение Задает точку входа процесса. Параметры • [in] `task` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует процесс. • [in] `entry` – точка входа начального потока процесса. • [out] `rc` – код возврата.	Создает условия для запуска кода, загруженного в память процесса.
`LoadElfSyms`	Назначение Загружает таблицу символов и таблицу строк из буферов MDL в память процесса.	Нет.

Буферы MDL содержат таблицу символов и таблицу строк из незагружаемых сегментов файла ELF. Эти таблицы нужны для получения данных обратной трассировки стека (сведений о стеках вызовов).

Параметры

- [in] `task` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует процесс.
- [in] `relocBase` – базовый адрес загрузки образа программы.
- [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.
- [in] `symSegAttr` – структура, содержащая параметры загрузки таблицы символов.
- [in] `symSize` – размер таблицы символов в байтах.
- [in] `strMdl` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует буфер MDL, содержащий таблицу строк.
- [in] `strSegAttr` – структура, содержащая параметры загрузки таблицы строк.
- [in] `strSize` – размер таблицы строк в байтах.
- [out] `rc` – код возврата.

`SetEnv`	Назначение Загружает параметры процесса в его память. Параметры • [in] `task` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует процесс. • [in] `env` – последовательность, содержащая параметры процесса. • [out] `rc` – код возврата.	Позволяет исчерпать память ядра, создавая в ней множество объектов.
`FreeSelfEnv`	Назначение Освобождает память текущего процесса, занятую параметрами, загруженными методом `SetEnv`. 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 текущий процесс.	Нет.

	Параметры • [in] `status` – код завершения текущего процесса. • [out] `rc` – код возврата.	
`Terminate`	Назначение Завершает 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 завершить другой процесс при наличии его дескриптора. (Маска прав дескриптора должна разрешать завершение процесса.)
`GetExitInfo`	Назначение Позволяет получить сведения о завершенном процессе. Параметры • [in] `task` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует завершенный процесс. • [out] `status` – код завершения процесса. • [out] `info` – объединение, содержащее сведения о завершенном процессе. • [out] `rc` – код возврата.	Нет.
`GetThreadContext`	Назначение	Позволяет нарушить изоляцию процесса, который перешел в "замороженное" состояние в результате необработанного исключения. Например, полученный контекст потока

	Позволяет получить контекст потока исполнения, входящего в процесс, который перешел в "замороженное" состояние в результате необработанного исключения. В "замороженном" состоянии исполнение процесса прекращается, но его ресурсы не освобождаются, чтобы можно было собрать данные об этом процессе. 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] `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` – последовательность, содержащая контекст потока исполнения. • [out] `rc` – код возврата.	исполнения может содержать значения переменных.
`GetNextVmRegion`	Назначение Позволяет получить сведения о регионе виртуальной памяти, принадлежащем процессу, который перешел в "замороженное" состояние в результате необработанного исключения.	Позволяет нарушить изоляцию процесса, который перешел в "замороженное" состояние в результате необработанного исключения. Изоляция нарушается, так как открывается доступ к региону памяти процесса.

	В "замороженном" состоянии исполнение процесса прекращается, но его ресурсы не освобождаются, чтобы можно было собрать данные об этом процессе. 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. Дескриптор идентифицирует буфер MDL, отображенный на регион виртуальной памяти. • [out] `rc` – код возврата.	
`TerminateAfterFreezing`	Назначение Завершает процесс, который перешел в "замороженное" состояние в результате необработанного исключения.	Позволяет завершить процесс, который перешел в "замороженное" состояние в результате необработанного исключения. Это не позволит собрать данные об этом процессе для диагностики.

	В "замороженном" состоянии исполнение процесса прекращается, но его ресурсы не освобождаются, чтобы можно было собрать данные об этом процессе. Процесс в "замороженном" состоянии не может быть повторно запущен, он может быть только завершен. 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.	
`GetName`	Назначение Позволяет получить имя текущего процесса. Параметры • [out] `name` – имя процесса. • [out] `rc` – код возврата.	Нет.
`GetPath`	Назначение Позволяет получить имя исполняемого файла, из которого запущен текущий процесс. Параметры • [out] `path` – имя исполняемого файла в ROMFS. • [out] `rc` – код возврата.	Нет.
`GetInitialThreadPriority`	Назначение Позволяет получить приоритет начального потока процесса. Параметры	Нет.

	• [in] `task` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор идентифицирует процесс. • [out] `priority` – значение, отражающее 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` – значение, задающее 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 приемник уведомлений, который настроен на получение уведомлений о завершении процессов.	Позволяет исчерпать память ядра, создавая в ней множество объектов.

	• [out] `strings` – последовательность, содержащая параметры процессов. • [out] `sids` – последовательность, содержащая идентификаторы безопасности процессов (SID каждого процесса). • [out] `count` – число процессов. • [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` – значение, задающее класс планирования начального потока исполнения. • [in] `priority` – значение, задающее приоритет начального потока исполнения. • [in] `params` – объединение, содержащее параметры класса планирования начального потока исполнения. • [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 исполнения). • 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).
`ReseedAslr`	Назначение Задает начальный вектор в генераторе случайных чисел для поддержки ASLR.	Нет.

	Влияет на результаты вызова метода `Allocate` службы виртуальной памяти в контексте заданного процесса. Параметры • [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 начальный вектор для генерации случайных чисел. • [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 текущего 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	Нет.

	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.	

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. • [out] `rc` – код возврата.	Нет.
`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	Нет.

	Позволяет получить число файлов в файловой системе. Параметры • [out] `count` – число файлов в файловой системе. • [out] `rc` – код возврата.	
`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. • [out] `name` – имя файла. • [out] `id` – уникальный идентификатор файла. • [out] `rc` – код возврата.	Нет.
`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.	Позволяет выполнить следующие действия: • Использовать образ ROMFS, содержащий произвольные программы и данные. • Получить доступ на чтение к некоторым объектам ядра.

	• [out] `rc` – код возврата.	

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. 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.	Позволяет установить системное время.

Служба слоя аппаратных абстракций

Служба предназначена для получения значений параметров HAL, работы с привилегированными регистрами, очистки кеша процессора, а также выполнения диагностического вывода.

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. • [in] `offset` – смещение регистра в диапазоне регистров. • [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 диапазона регистров и смещение регистра в этом диапазоне.
`SetPrivRegRange`	Назначение Задает значение привилегированного регистра. Параметры • [in] `regRange` – имя диапазона регистров.	Позволяет выполнить следующие действия: • Задать значение привилегированного регистра. • Организовать канал передачи данных с процессом, который имеет доступ к методу `GetPrivReg` или `GetPrivRegRange`.

	• [in] `offset` – смещение регистра в диапазоне регистров. • [in] `val` – значение регистра. • [out] `rc` – код возврата.	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 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.	Позволяет заполнить диагностический вывод фиктивными данными (например, неинформативными).

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	Потенциальная опасность метода

	параметры метода	
`Start`	Назначение Включает отладочный режим контроллера XHCI. Параметры • [out] `rc` – код возврата.	Позволяет настроить контроллер XHCI, чтобы диагностический вывод выполнялся через порт USB (версии 3.0 или более поздней).
`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	Нет.

	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. • [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 пользовательского кода и кода ядра, получения сведений о покрытии кода ядра и пользовательского кода, а также 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
`CreateUser`	Назначение Назначает профилирование пользовательского кода. Результатом профилирования является статистика исполнения пользовательского кода в контексте заданного потока исполнения. Статистика показывает, сколько раз за время профилирования сработал пользовательский код из разных участков заданного диапазона виртуальных адресов. Параметры	Позволяет исчерпать память ядра, создавая в ней множество объектов.

	• [in] `tid` – идентификатор потока исполнения (TID). • [in] `from` – начальный адрес диапазона виртуальных адресов, для которого собирается статистика. • [in] `to` – конечный адрес диапазон виртуальных адресов, для которого собирается статистика. • [in] `scale` – значение, определяющее гранулярность разбиения пользовательского кода внутри диапазона виртуальных адресов, заданного через параметры `from` и `to`. От этого значения зависит, на сколько участков будет разбит диапазон адресов. • [out] `rc` – код возврата.	
`DestroyUser`	Назначение Отменяет профилирование пользовательского кода. Параметры • [in] `tid` – идентификатор потока исполнения (TID). • [out] `rc` – код возврата.	Нет.
`CreateKernel`	Назначение Назначает профилирование кода ядра. Результатом профилирования является статистика исполнения кода ядра. Статистика показывает, сколько раз за время профилирования сработал код ядра из разных участков диапазона адресов памяти процесса, вызвавшего этот метод. Диапазон виртуальных адресов, занятых кодом ядра, идентичен для всех процессов. Статистика исполнения кода ядра собирается в общем, а не в контексте одного процесса или потока исполнения. Параметры • [out] `from` – начальный адрес диапазона виртуальных адресов, для которого собирается статистика. • [out] `to` – конечный адрес диапазон виртуальных адресов, для которого собирается статистика. • [out] `scale` – значение, отражающее гранулярность разбиения кода ядра внутри диапазона виртуальных адресов, соответствующего параметрам `from` и `to`. Это значение зависит от того, на сколько участков разбит диапазон адресов. • [out] `size` – размер данных, содержащих статистику, в байтах.	Нет.

	• [out] `rc` – код возврата.	
`DestroyKernel`	Назначение Отменяет профилирование кода ядра. Параметры • [out] `rc` – код возврата.	Нет.
`StartKernel`	Назначение Запускает профилирование кода ядра. Параметры • [out] `rc` – код возврата.	Нет.
`StopKernel`	Назначение Останавливает профилирование кода ядра. Параметры • [out] `rc` – код возврата.	Нет.
`GetKernelData`	Назначение Позволяет получить данные, содержащие статистику исполнения кода ядра, полученную в результате профилирования. Параметры • [in] `buf` – указатель на буфер для сохранения данных, содержащих статистику исполнения кода ядра. • [out] `rc` – код возврата.	Нет.
`GetCoverageData`	Назначение Позволяет получить сведения о покрытии кода ядра. Параметры • [in] `index` – индекс для перечисления объектных файлов, содержащих инструментированный код для сбора покрытия. Нумерация начинается с нуля. • [out] `buf` – последовательность, содержащая сведения о покрытии кода объектного файла (в формате `gcda`).	Нет.

	• [out] `size` – размер данных, содержащих сведения о покрытии кода объектного файла, в байтах. • [out] `name` – путь к файлу `*.gcda`, назначенный при компиляции. • [out] `rc` – код возврата.	
`FlushGcov`	Назначение Выводит сведения о покрытии кода ядра 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` – путь к файлу `*.gcda`, назначенный 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.

Служба управления памятью для ввода-вывода

Служба предназначена для управления изоляцией регионов физической памяти, используемых устройствами на шине PCIe. (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 текущим процессом. Параметры • [in] `bdf` – адрес устройства на шине PCIe формате BDF. • [out] `rc` – код возврата.	Позволяет прикрепить устройство на шине PICIe, управляемое другим процессом, к домену IOMMU, ассоциированному с текущим процессом, что приведет к неработоспособности устройства. Рекомендуется контролировать адрес устройства на шине PCIe.
`Detach`	Назначение Открепляет устройство на шине PCIe от домена IOMMU, ассоциированного с текущим 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 принятия запроса сервером 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. • [out] `rc` – код возврата.	Позволяет создать нагрузку на сервер, отправляя множество запросов на создание IPC-канала.
`Listen`	Назначение Проверяет наличие запроса 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 IPC-дескриптором. • [out] `handle` – значение, двоичное представление которого состоит из нескольких полей, включая поле дескриптора и поле маски прав дескриптора. Дескриптор является серверным IPC-дескриптором. • [out] `rc` – код возврата.	Нет.

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.	Позволяет выключить и включить процессоры.

	Параметры • [in] `request` – значение, задающее множество процессоров в активном состоянии. • [in] `timeout` – время ожидания выполнения запроса в миллисекундах. • [out] `rc` – код возврата.	
`GetCpusOnline`	Назначение Позволяет получить сведения о том, какие процессоры находятся в активном состоянии. Параметры • [out] `online` – значение, отражающее множество процессоров в активном состоянии. • [out] `rc` – код возврата.	Нет.

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.	Позволяет исчерпать память ядра, создавая в ней множество объектов.

	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". Используется для идентификации записи в полученных уведомлениях. • [out] `rc` – код возврата.	
`UnsubscribeFromEvent`	Назначение Удаляет из приемника уведомления, которые соответствуют записи вида "ресурс – маска событий" с заданным идентификатором. 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 из приемника уведомления, которые соответствуют заданному ресурсу. 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] `evId` – идентификатор записи вида "ресурс – маска событий", соответствующей ресурсу, для которого извлечены уведомления. • [out] `evMask` – маска событий, произошедших с ресурсом. • [out] `rc` – код возврата.	Нет.
`DropAndWake`	Назначение Удаляет из заданного приемника уведомлений все записи вида "ресурс – маска событий"; возобновляет исполнение всех потоков, ожидающих наступления события, ассоциированного с заданным приемником уведомлений; 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	Нет.

	• [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.	

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	Позволяет освободить значения, используемые другими процессами в REE в качестве уникальных

	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.	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.	Позволяет получить доступ к произвольному региону физической памяти REE из TEE.

	• [in] `pages` – размер региона физической памяти в страницах памяти. • [out] `rc` – код возврата.	

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()` во всех потоках текущего процесса. `Call()` может быть прерван только тогда, когда он ожидает вызова `Recv()` сервером. `Recv()` может быть прерван только тогда, когда он ожидает получения данных 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.	Нет.

	• [out] `rc` – код возврата.	
`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	Нет.

	• [in] `cpuGroupId` – индекс процессорной группы. Нумерация начинается с нуля. • [out] `oppId` – индекс текущего OPP. Нумерация начинается с нуля. • [out] `rc` – код возврата.	
`SetOppId`	Назначение Устанавливает заданный OPP для заданной процессорной группы. Параметры • [in] `GroupId` – индекс процессорной группы. Нумерация начинается с нуля. • [in] `oppId` – индекс OPP. Нумерация начинается с нуля. • [out] `rc` – код возврата.	Позволяет изменить частоту процессорной группы.

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

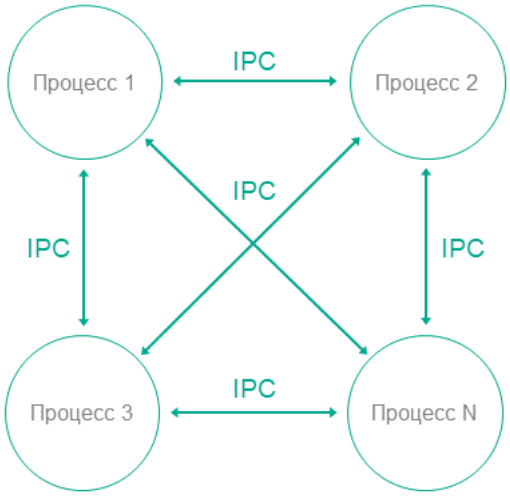
Наивная реализация приложения объединяет множество разнородных по необходимым привилегиям функций в одном компоненте, вынуждая его запускаться с максимальным из необходимых уровней привилегий.

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:

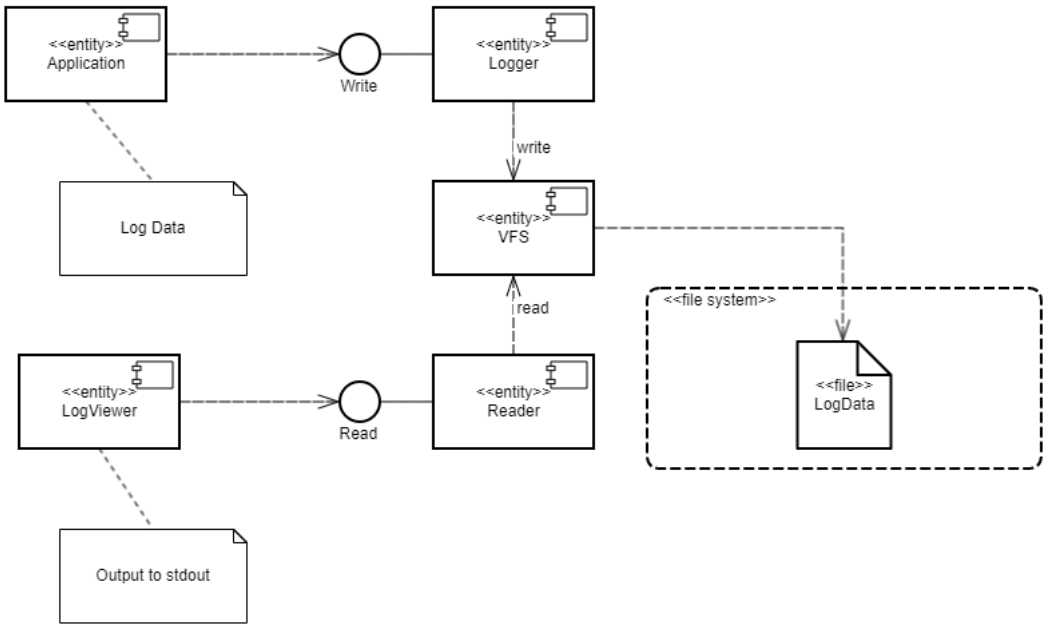
- генерация информации для записи в журнал;
- сохранение информации в журнал;
- чтение записей из журнала;
- предоставление записей в удобном для потребителя виде.

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.

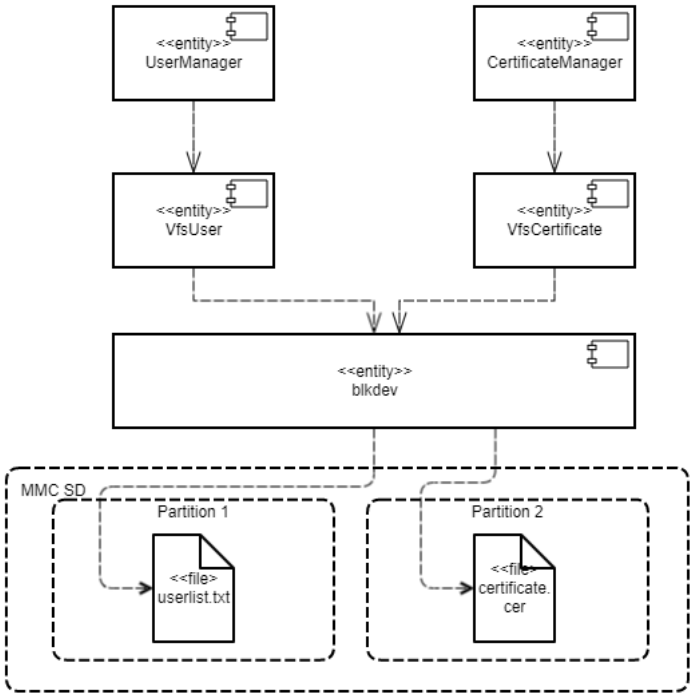
Архитектура примера

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
hdd.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:

- 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 `embed_ext2_with_separate_vfs` 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

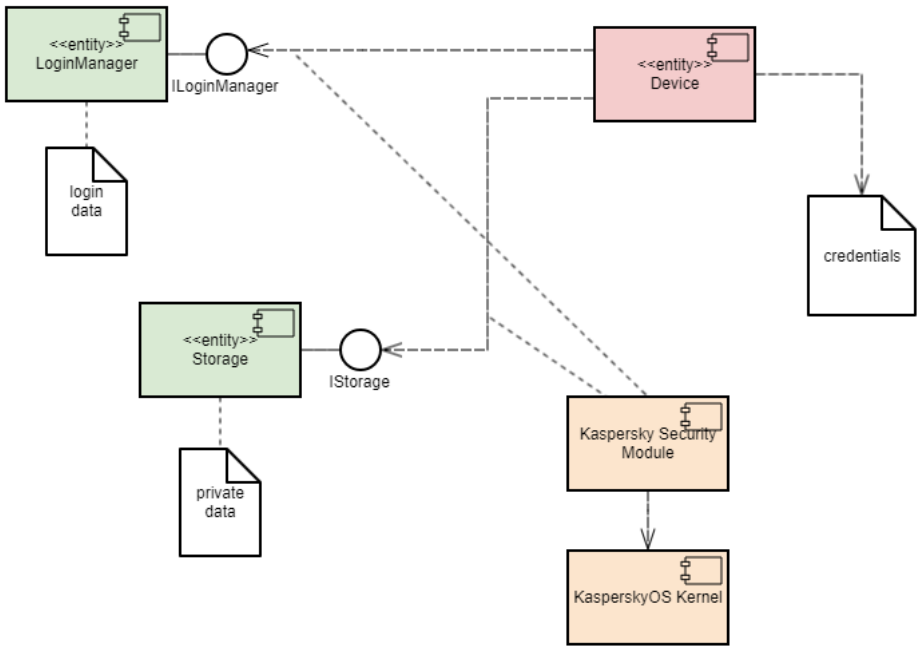
Пример `Device Access` демонстрирует использование паттерна Privilege Separation.

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

Политика безопасности должна разделять данные 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. Откройте файл скрипта `cross-build.sh` в текстовом редакторе.
4. Добавьте в скрипт флаг сборки `-D RUN_TESTS="y" \` (например, после флага сборки `-D CMAKE_BUILD_TYPE:STRING=Release \`).
5. Сохраните файл скрипта, а затем выполните команду:

   ```
   $ sudo ./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".

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

```
#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 из него 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

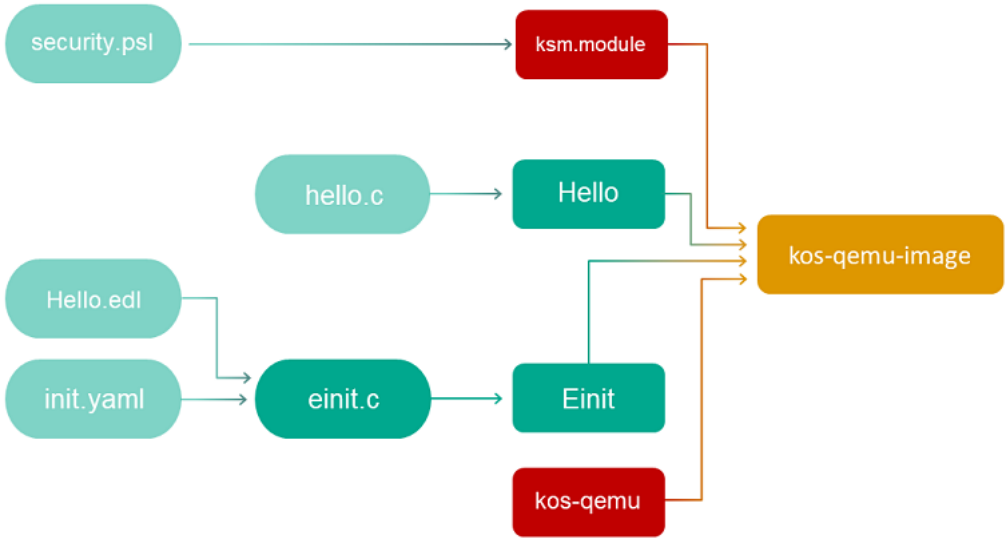
Код примера и скрипты для сборки находятся по следующему пути:

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

Сборка и запуск примера

См. "Сборка и запуск примеров".

Общая схема сборки примера hello выглядит следующим образом:



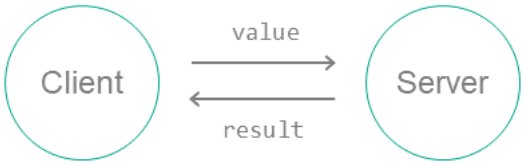
Пример echo

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.

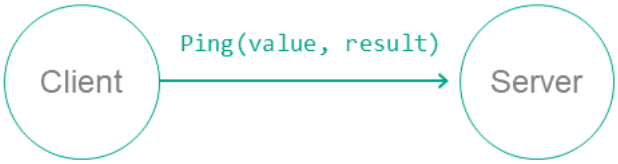


Чтобы организовать такое взаимодействие программ, потребуется:

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
```

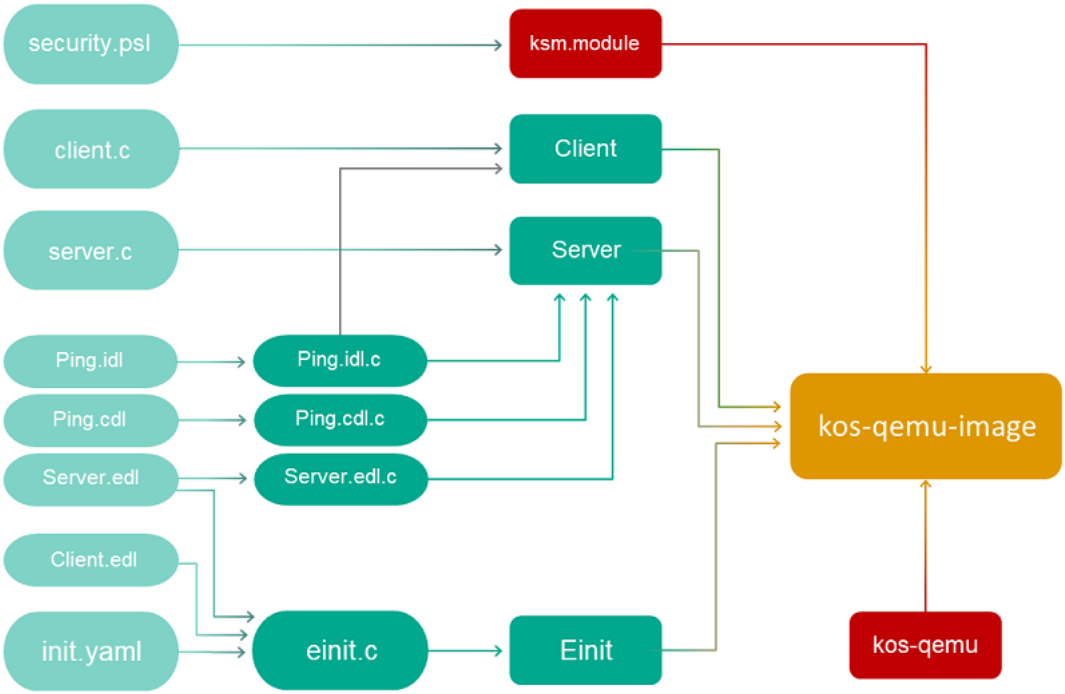
Пример echo состоит из следующих исходных файлов:

- `client/src/client.c` – реализация программы `Client`;
- `server/src/server.c` – реализация программы `Server`;
- `resources/Server.edl`, `resources/Client.edl`, `resources/Ping.cdl`, `resources/Ping.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 две программы: `Client` и `Server`.

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 `Failed to call...`

> 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 всех программ и позволяет любой программе обращаться к программам `Core` и `Server`. При этом обращениями к программе `Server` управляют методы модели безопасности Flow.

Конечный автомат, описанный в конфигурации объекта `request_state` модели безопасности Flow, имеет два состояния: `ping_next` и `pong_next`. Исходное состояние – `ping_next`. Разрешены только переходы из `ping_next` в `pong_next` и обратно.

При вызове методов `Ping` и `Pong` проверяется текущее состояние объекта `request_state`. В состоянии `ping_next` разрешен только вызов `Ping`, при этом состояние изменится на `pong_next`. Аналогично, в состоянии `pong_next` разрешен только вызов `Pong`, при этом состояние изменится на `ping_next`.

Таким образом, методы `Ping` и `Pong` разрешено вызывать только по очереди.

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._

/* Создание объекта модели безопасности Flow */
policy object request_state : Flow {
    type States = "ping_next" | "pong_next"
    config = {
        states      : ["ping_next" , "pong_next"],
        initial     : "ping_next",
        transitions : {
            "ping_next" : ["pong_next"],
            "pong_next" : ["ping_next"]
        }
    }
}

/* Запуск любых программ разрешен. */
execute {
    grant ()
}

/* Любые запросы разрешены. */
request {
    grant ()
}

/* Любые ответы разрешены. */
response {
    grant ()
}

/* Включение EDL-файлов */
use EDL kl.core.Core
use EDL ping.Client
use EDL ping.Server
use EDL Einit

/* При запуске программы Server инициировать эту программу с конечным автоматом */
execute dst=ping.Server {
    request_state.init {sid: dst_sid}
}

/* При вызове метода Ping проверить, что конечный автомат находится в состоянии
ping_next.
   Если это так, разрешить вызов метода Ping и перевести конечный автомат в состояние
pong_next. */
```

```
request dst=ping.Server, endpoint=controlimpl.connectionimpl, method=Ping {
    request_state.allow {sid: dst_sid, states: ["ping_next"]}
    request_state.enter {sid: dst_sid, state: "pong_next"}
}

/* При вызове метода Pong проверить, что конечный автомат находится в состоянии
pong_next.
   Если это так, разрешить вызов метода Pong и перевести конечный автомат в состояние
ping_next. */
request dst=ping.Server, endpoint=controlimpl.connectionimpl, method=Pong {
    request_state.allow {sid: dst_sid, states: ["pong_next"]}
    request_state.enter {sid: dst_sid, state: "ping_next"}
}
```

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`.

Этот пример не содержит реализации драйверов блочных устройств, с которыми работает `Client`. Эти драйверы (программы `ATA` и `SDCard`) поставляются в составе KasperskyOS Community Edition и добавляются в файле сборки `./CMakeLists.txt`.

Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

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

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
hdd.img -kernel kos-qemu-image
```

Также см. "Сборка и запуск примеров".

Пример embed_ext2_with_separate_vfs

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/embed_ext2_with_separate_vfs
```

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
hdd.img -kernel kos-qemu-image
```

Также см. "Сборка и запуск примеров".

Подготовка SD-карты для запуска на Raspberry Pi 4 B

Для запуска примера `embed_ext2_with_separate_vfs` 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 `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.

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 используются программы `VfsRamfs` и `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 пример входят следующие файлы конфигурации:

- `./resources/include/config.h.in` содержит описание бэкенда файловой системы, которая будет использоваться в решении: `sdcard` или `ramfs`.

  Для каждого бэкенда в решении также используется отдельная программа VFS: `VfsSdCardFs` или `VfsRamfs` соответственно.

- Директории `./resources/ramfs/etc` и `/resources/sdcard/etc` содержат файлы конфигурации для программ VFS и `Ntpd`. Для конфигурации программы `ntpd` используется стандартный синтаксис `ntpd.conf`.

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

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".

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. 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 используются программы `VfsRamfs` и `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 пример входят следующие файлы конфигурации:

- `./resources/include/config.h.in` содержит описание бэкенда файловой системы, которая будет использоваться в решении: `sdcard` или `ramfs`.

  Для каждого бэкенда в решении также используется отдельная программа VFS: `VfsSdCardFs` или `VfsRamfs` соответственно.

- Директории `./resources/ramfs/etc` и `/resources/sdcard/etc` содержат файлы конфигурации для программы VFS.

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

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".

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 используются программы `VfsRamfs` и `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

В пример входят следующие файлы конфигурации:

- `./resources/include/config.h.in` содержит описание бэкенда файловой системы, которая будет использоваться в решении: `sdcard` или `ramfs`.

  Для каждого бэкенда в решении также используется отдельная программа VFS: `VfsSdCardFs` или `VfsRamfs` соответственно.

- Директории `./resources/ramfs/etc` и `/resources/sdcard/etc` содержат файлы конфигурации для программ VFS и `Dhcpcd`. Для конфигурации программы `dhcpcd` используется стандартный синтаксис `dhcpcd.conf`.

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

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".

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. 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 используются программы `VfsRamfs` и `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 пример входят следующие файлы конфигурации:

- `./resources/include/config.h.in` содержит описание бэкенда файловой системы, которая будет использоваться в решении: `sdcard` или `ramfs`.

  Для каждого бэкенда в решении также используется отдельная программа VFS: `VfsSdCardFs` или `VfsRamfs` соответственно.

- Директории `./resources/ramfs/etc` и `/resources/sdcard/etc` содержат файлы конфигурации для программ `VFS`, `Dhcpcd` и `Ntpd`.

Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

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

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".

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. 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 используются программы `VfsRamfs` и `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 пример входят следующие файлы конфигурации:

- `./resources/include/config.h.in` содержит описание бэкенда файловой системы, которая будет использоваться в решении: `sdcard` или `ramfs`.

  Для каждого бэкенда в решении также используется отдельная программа VFS: `VfsSdCardFs` или `VfsRamfs` соответственно.

- Директории `./resources/ramfs/etc` и `/resources/sdcard/etc` содержат файлы конфигурации для программ `VFS`, `Dhcpcd` и `Ntpd`.

Файлы примера

Код примера и скрипты для сборки находятся по следующему пути:

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

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".

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 используются программы `VfsRamfs` и `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".

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".

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 используются программы `VfsRamfs` и `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".

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_NET_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".

Лицензирование программы

Условия использования программы изложены в лицензионном договоре или подобном документе, на основании которого используется программа.

Предоставление данных

KasperskyOS Community Edition не запрашивает, не хранит и не обрабатывает никакую персональную информацию, а также никакую другую информацию, не относящуюся к персональным данным.

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.

CentOS – товарный знак компании Red Hat, Inc.

Debian – зарегистрированный товарный знак Software in the Public Interest, Inc.

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.

Intel и Core – товарные знаки Intel Corporation, зарегистрированные в Соединенных Штатах Америки и в других странах.

Linux – товарный знак Linus Torvalds, зарегистрированный в США и в других странах.

Raspberry Pi – товарный знак Raspberry Pi Foundation.

Ubuntu является зарегистрированным товарным знаком Canonical Ltd.

Visual Studio, Windows являются товарными знаками группы компаний Microsoft.