kaspersky

KasperskyOS Community Edition 1.3

© 2025 AO Kaspersky Lab

Contents

- What's new
- Introduction
- About KasperskyOS Community Edition
  - Distribution kit
  - System requirements
  - Included third-party libraries and applications
  - Limitations and known issues
  - Critical changes in version 1.3
- Overview of KasperskyOS
  - Overview
  - KasperskyOS architecture
  - IPC
    - IPC mechanism
    - IPC control
    - Transport code for IPC
    - IPC between a process and the kernel
  - Resource Access Control
  - Structure and startup of a KasperskyOS-based solution
- The KasperskyOS Developer's Quick Start Guide
- Getting started
  - Using a Docker container
  - Installation and removal
  - Configuring the development environment
  - KasperskyOS SDK Extension for Visual Studio Code
    - Installing and removing the extension
    - Extension functions
      - Starting a basic KasperskyOS-based solution image
      - Building programs in Visual Studio Code
      - Starting programs in a basic solution image
      - Debugging a program that is part of a KPA package
        - Connecting a debugger when starting a program in a basic image
        - Connecting a debugger while a program is running from a basic image
    - Extension settings
  - Building and running examples
    - Building the examples
    - Running examples on QEMU
    - Preparing Raspberry Pi 4 B to run examples
    - Preparing Radxa ROCK 3A to run examples
    - Running examples on Raspberry Pi 4 B or Radxa ROCK 3A
- Development for KasperskyOS
  - Starting processes
    - Overview: Einit and init.yaml
    - Example init descriptions
    - Starting processes using the system program ExecutionManager
    - Overview: Env program
    - Examples of using Env to set the startup parameters and environment variables of programs

- File systems and network
  - Contents of the VFS component
  - Creating an IPC channel to VFS
  - Including VFS functionality in a program
  - Overview: startup parameters and environment variables of VFS
  - Mounting file systems when VFS starts
  - Using VFS backends to separate data streams
  - Creating a VFS backend
  - Dynamically configuring the network stack
- IPC and transport
  - Creating IPC channels
  - Adding an endpoint from KasperskyOS Community Edition to a solution
  - Creating and using your own endpoints
    - Overview: IPC message structure
    - Example generation of transport methods and types
    - Working with an IPC message arena
    - Transport code in C++
      - Statically creating IPC channels for C++ development
      - Dynamically creating IPC channels for C++ development
- Working with KPA packages
  - KPA package manifest
    - Application object
    - Platform object
    - List of "components" objects
    - List of runConfiguration objects
    - PrivateStorage object
  - Tools for managing KPA packages
    - cas-pack tool
    - cas-inspect tool
    - cas-pm tool
  - PackageManager component usage scenario
- KasperskyOS API
  - Return codes
  - libkos library
    - Managing handles (handle_api.h)
      - Handle permissions mask
      - Creating handles
      - Transferring handles
      - Duplicating handles
      - Dereferencing handles
      - Revoking handles
      - Closing handles
      - Getting a security ID (SID)
      - OCap usage example
    - Managing virtual memory (vmm_api.h)
      - Allocating and freeing memory
      - Creating shared memory
      - Preparing ELF image segments to be loaded into process memory

- Allocating and freeing memory (alloc.h)
- Using DMA (dma.h)
- Memory-mapped I/O (mmio.h)
- Managing interrupt processing (irq.h)
- Managing threads (high-level API thread.h)
- Managing threads (low-level API thread_api.h)
- Managing processes (high-level API task.h)
- Managing processes (low-level API task_api.h)
- Initializing IPC transport for interprocess communication and managing IPC request processing (transport-kos.h, transport-kos-dispatch.h)
- Initializing IPC transport for querying the security module (transport-kos-security.h)
- Generating random numbers (random_api.h)
- Getting and changing time values (time_api.h)
- Using notifications (notice_api.h)
- Dynamically creating IPC channels (cm_api.h, ns_api.h)
- Using synchronization primitives (event.h, mutex.h, rwlock.h, semaphore.h, condvar.h)
- Managing I/O memory isolation (iommu_api.h)
- Using queues (queue.h)
- Using memory barriers (barriers.h)
- Executing system calls (syscalls.h)
- IPC interrupt (ipc_api.h)
- Using sessions (session.h)
- Using KosObjects (objects.h)
- Using KosObject containers (objcontainer.h)
- Using KosStrings (strings.h)
- KasperskyOS kernel XHCI DbC driver management (xhcidbg_api.h)
- Receiving security audit data (vlog_api.h)
- Using futexes (sync.h)
- Getting IPC handles and RIIDs to use statically created IPC channels (sl_api.h)
- Power management (pm_api.h)

POSIX support

- POSIX support limitations
- POSIX implementation specifics
- Concurrently using POSIX and the libkos API

Obtaining statistical data on the system through the libc library API

Dynamically creating IPC channels using the DCM system program

- Using the base API (dcm_api.h)
- Using the additional server API for servers (pub.h)
- Using the additional API for clients (subscr.h)

MessageBus component

- IProviderFactory interface
- IProviderControl interface
- IProvider interface (MessageBus component)
- ISubscriber, IWaiter and ISubscriberRunner interfaces

LogRR component

- Adding a log server and output channels to a solution
- Getting log entries when running in QEMU
- Using macros to send messages to a log

- Advanced capabilities when sending messages to a log
- Log levels
- LogRR program API
  - logrr_clog library
    - ClogLog() function and CLOG macro
    - LogrrLogLevel enumerated type
  - logrr_cpp library
    - Log() method and LOG macro
    - LogLevel class
    - LogIface class
- ExecutionManager component
  - IProcessControl interface
  - IApplicationController interface
  - IStateProvider interface
  - ISystemController interface
- PackageManager component
  - IPackageController interface
  - IPackageManifest interface

Building a KasperskyOS-based solution
- Building a solution image
- Build process overview
- Using CMake from the contents of KasperskyOS Community Edition
  - CMakeLists.txt root file
  - CMakeLists.txt files for building applications
  - CMakeLists.txt file for building the Einit program
  - init.yaml.in template
  - security.psl.in template
- CMake libraries in KasperskyOS Community Edition
  - platform library
  - nk library
    - generate_edl_file()
    - nk_build_idl_files()
    - nk_build_cdl_files()
    - nk_build_edl_files()
    - Generating transport code for development in C++
      - add_nk_idl()
      - add_nk_cdl()
      - add_nk_edl()
  - image library
    - build_kos_qemu_image()
    - build_kos_hw_image()
  - build_sd_image library
  - kpa library
    - add_kpa_package()
    - add_kpa_component()
- Building without CMake
  - Tools for building a solution
    - Build scripts and tools

nk-gen-c
nk-psl-gen-c
einit
makekss
makeimg
Cross compilers
Using dynamic libraries
Prerequisites for using dynamic libraries
Life cycle of a dynamic library
Including the BlobContainer system program in a KasperskyOS-based solution
Building dynamic libraries
Adding dynamic libraries to a KasperskyOS-based solution
Debugging programs in a KasperskyOS-based solution
Debugging programs using the QEMU GDB server
Preparations for debugging
Initial steps of debugging
Special considerations for debugging emulated hardware platforms with SMP
Debugging programs using the GDB server of the KasperskyOS kernel
Preparations for debugging in QEMU
Initial steps of debugging in QEMU
Preparations for debugging on the hardware platform
Initial steps of debugging on the hardware platform
Working with processes and threads
Analyzing stack backtrace data when a process abnormally terminates
Developing security policies
Formal specifications of KasperskyOS-based solution components
Names of process classes, components, packages and interfaces
EDL description
CDL description
IDL description
IDL data types
Integer expressions in IDL
Describing a security policy for a KasperskyOS-based solution
General information about a KasperskyOS-based solution security policy description
PSL language syntax
Setting the global parameters of a KasperskyOS-based solution security policy
Including PSL files in a KasperskyOS-based solution security policy description
Including EDL files in a KasperskyOS-based solution security policy description
Creating security model objects
Binding methods of security models to security events
Creating security audit profiles
Creating and performing tests for a KasperskyOS-based solution security policy
PSL data types
Examples of binding security model methods to security events
Example descriptions of basic security policies for KasperskyOS-based solutions
Examples of security audit profiles
Examples of tests for KasperskyOS-based solution security policies
KasperskyOS security models

Pred security model
Bool security model
Math security model
Struct security model
Base security model
Regex security model
HashSet security model
HashSet security model object
HashSet security model init rule
HashSet security model fini rule
HashSet security model add rule
HashSet security model remove rule
HashSet security model contains expression
StaticMap security model
StaticMap security model object
StaticMap security model init rule
StaticMap security model fini rule
StaticMap security model set rule
StaticMap security model commit rule
StaticMap security model rollback rule
StaticMap security model get expression
StaticMap security model get_uncommitted expression
Flow security model
Flow security model object
Flow security model init rule
Flow security model fini rule
Flow security model enter rule
Flow security model allow rule
Flow security model query expression
Mic security model
Mic security model object
Mic security model create rule
Mic security model delete rule
Mic security model execute rule
Mic security model upgrade rule
Mic security model call rule
Mic security model invoke rule
Mic security model read rule
Mic security model write rule
Mic security model query_level expression
Using templates when creating a KasperskyOS-based solution security policy description
Methods of KasperskyOS core endpoints
Virtual memory endpoint
I/O endpoint
Threads endpoint
Handles endpoint
Processes endpoint
Synchronization endpoint

- File system endpoints
- Time endpoint
- Hardware abstraction layer endpoint
- KasperskyOS kernel XHCI DbC driver management endpoint
- Audit endpoint
- Profiling endpoint
- I/O memory isolation management endpoint
- Connections endpoint
- Power management endpoint
- Notifications endpoint
- IPC interrupt endpoint
- Hardware platform firmware communication endpoint
- Using the system programs Klog and KlogStorage to perform a security audit
  - Example of adding the system program Klog to a solution
  - Example of adding the system program KlogStorage to a solution to forward audit data to standard error
  - Example of adding the system program KlogStorage to a solution to write audit data to a file
- Security patterns for development under KasperskyOS
  - Distrustful Decomposition pattern
    - Secure Logger example
    - Separate Storage example
  - Defer to Kernel pattern
    - Defer to Kernel example
  - Policy Decision Point pattern
  - Privilege Separation pattern
    - Device Access example
  - Information Obscurity pattern
    - Secure Login (Civetweb, TLS-terminator) example
- Developing drivers for KasperskyOS
- Appendices
  - Examples in KasperskyOS Community Edition
    - hello example
    - echo example
    - ping example
    - net_with_separate_vfs example
    - net2_with_separate_vfs example
    - embedded_vfs example
    - vfs_extfs example
    - multi_vfs_ntpd example
    - multi_vfs_dns_client example
    - multi_vfs_dhcpcd example
    - mqtt_publisher (Mosquitto) example
    - mqtt_subscriber (Mosquitto) example
    - gpio_input example
    - gpio_output example
    - gpio_interrupt example
    - gpio_echo example
    - koslogger example
    - pcre example

- messagebus example
- i2c_ds1307_rtc example
- i2c_bm8563_rtc example
- iperf_separate_vfs example
- uart example
- spi_check_regs example
- barcode_scanner example
- watchdog_system_reset example
- shared_libs example
- pal_tests example
- hello_from_rust example
- hello_corrosion example
- mass_storage example
- can_loopback example
- can2can example
- adc_hello example
- Information about certain limits set in the system

Licensing

Data provision

Glossary

- Address Space Layout Randomization
- Affinity mask
- Application
- Arena chunk descriptor
- Arena descriptor
- ASLR
- Callable handle
- Capability
- CDL
- Client
- Client library of the solution component
- Client process
- Conditional variable
- Constant part of an IPC message
- Critical section
- DCM handle
- Describing a security policy for a KasperskyOS-based solution
- Direct memory access
- DMA
- DMA buffer
- EDL
- Empty process
- Endpoint
- Endpoint interface
- Endpoint method
- Endpoint method ID
- Event
- Event mask

Execute interface
Formal specification of the KasperskyOS-based solution component
Frozen state of a process
Futex
Global system information
GSI
Handle
Handle dereferencing
Handle inheritance tree
Handle permissions mask
Handle transport container
Hardware interrupt
IDL
Inferior
Init description
Initializing program
Interface method
Interprocess communication
IPC
IPC channel
IPC handle
IPC message
IPC message arena
IPC request
IPC response
IPC transport
KasperskyOS
KasperskyOS security model
KasperskyOS-based solution
KasperskyOS-based solution component
KPA package
KPA package manifest
KSM
KSS
Listener handle
MDL
MDL buffer
Memory barrier
Memory Descriptor List
Message signaled interrupt
MID
MMIO
MMIO memory
Mutex
Notification receiver
OCap
Operating performance point
PAL

PCB
Process
Process control block
Processor mask
Program
PSL
Read-write lock
Recursive mutex
Resource
Resource consumer
Resource integrity level
Resource provider
Resource transfer context
Resource transfer context object
RIID
Runtime Implementation Identifier
SCP
Security audit
Security audit configuration
Security audit data
Security audit profile
Security audit runtime-level
Security context
Security event
Security ID
Security interface
Security model expression
Security model method
Security model object
Security model rule
Security module decision
Security pattern
Security pattern system
Security policy for a KasperskyOS-based solution
Security template
Seed
Semaphore
Server
Server library of the solution component
Server process
SID
SMP
Static connection page
Subject integrity level
System program
System resource
TCB
Thread

- Thread control block
- Thread local storage
- TLS
- Transport code
- Transport library
- User resource
- User resource context

Information about third-party code

Trademark notices

What's new

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

> Due to modifications made to SDK components, you must make changes to the application code that was developed using KasperskyOS Community Edition version 1.2 before using that code with KasperskyOS Community Edition version 1.3. For more details, refer to Critical changes in version 1.3.

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

- google/re2 (2022-02-01)
- rust (1.59)
- wpa_supplicant (2.10)

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

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

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

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

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

- Updated the following third-party libraries and applications:
    - FFmpeg

- libxml2
- Eclipse Mosquitto
- opencv
- OpenSSL
- protobuf
- sqlite
- usb

- Added support for the Raspberry Pi 4 Model B hardware platform (Revision 1.5).

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

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

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

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

Introduction

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

- How-to guides that describe the actions that a developer should take to get specific practical results.

  For these purposes, developers can use the following:

  - API functions and macros (in the C language)
  - Shell commands, GDB commands, and CMake commands
  - Specialized languages used to automatically generate source code

  A how-to guide may be presented as an example of code with explanations.

- Reference guides:

  - API information
  - Syntax of shell commands, GDB commands, and CMake commands
  - Syntax of specialized languages used to automatically generate source code
  - Program startup parameters
  - Environment variables of programs
  - POSIX support details
  - Information on the KasperskyOS kernel methods used for a security policy description

- KasperskyOS conceptual guides.

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

Contents of Guide sections

Section name	Section contents
About KasperskyOS Community Edition	General information about the SDK, including the system requirements for SDK installation.
Overview of KasperskyOS	KasperskyOS conceptual guides: • KasperskyOS architecture • Kaspersky Security System technology • Generating source code from code in specialized languages • Managing interaction between different processes and between processes and the KasperskyOS kernel • Resource Access Control • KasperskyOS boot image contents
The KasperskyOS Developer's Quick Start Guide	• Practical Guide to developing KasperskyOS solutions with concise explanations and links to other sections for a quick KasperskyOS development setup

Getting started	Practical guides required for getting started with the SDK: • Using a Docker container to install the SDK • Installing and uninstalling the SDK • Configuring the development environment • Using the KasperskyOS SDK Extension for Visual Studio Code • Building and running examples
Development for KasperskyOS	How-to guides: • Starting processes • Using file systems and network stack • Setting up interprocess communication • Working with KPA packages Reference guides: • Syntax of a specialized language used to configure program startup parameters and environment variables, and the topology for interprocess communication • Startup settings and environment variables for the system program that implements the file systems and network stack
KasperskyOS API	How-to guides: • Using macros designed for working with error codes in system software and applications • Using APIs provided by the libkos library (basic library of KasperskyOS), including capability-based access control • Using the API of the system program that provides dynamic configuration for the IPC topology • Using the MessageBus program to implement the message bus • Using the LogRR program for logging • Using the ExecutionManager component for process management • Using the PackageManager component for KPA packages management Reference guides: • Information on return codes for system software and applications • Information on APIs provided by the libkos library • POSIX support details • Information about the API of the system program that provides dynamic configuration for the IPC topology
Building a KasperskyOS-based solution	How-to guides: • Build process overview • Building with CMake • Using dynamic libraries Reference guides: • Syntax of macros applied during a build to generate constructs in a specialized language used to configure program startup parameters and environment variables, and the topology for interprocess communication • Syntax of a macro applied during a build to generate constructs in a specialized language used for a security policy description

	• Syntax of CMake commands from CMake libraries provided in the SDK • Syntax of shell commands for running build scripts and tools
Debugging programs in a KasperskyOS-based solution	How-to guides: • Debugging using the QEMU GDB server • Debugging using the GDB server of the KasperskyOS kernel in QEMU and on the hardware platform • Analyzing stack backtrace data when a process abnormally terminates Reference guides on the syntax of GDB commands for working with processes and threads
Developing security policies	How-to guides: • Creating specifications for programs that run in KasperskyOS • Describing a security policy • Testing a security policy • Logging a security policy Reference guides: • Syntax of specialized languages used to create specifications for programs that run in KasperskyOS • Syntax of a specialized language used to describe a security policy (including KasperskyOS security models) • Syntax of a specialized language used to create security policy tests • Information on the KasperskyOS kernel methods used for a security policy description
Security patterns for development under KasperskyOS	Conceptual guides on the types of risks and security threats during software development, and a description of architectural patterns that can be employed to enhance the security of a solution.
Appendices	List and description of examples provided in the SDK Reference guides on certain limits set in the system

About KasperskyOS Community Edition

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

KasperskyOS Community Edition will help you:

- Learn the principles and techniques of "secure by design" development based on practical examples.
- Explore KasperskyOS as a potential platform for implementing your own projects.
- Make prototypes of solutions (primarily Embedded/IoT) based on KasperskyOS.
- Port applications/components to KasperskyOS.
- Explore security issues in software development.

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

You can download KasperskyOS Community Edition here.

In addition to this documentation, we also recommend that you explore the materials provided in the specific KasperskyOS website section for developers.

Distribution kit

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

The distribution kit of KasperskyOS Community Edition includes the following:

- Deb package for installation of KasperskyOS Community Edition, including:
  - Image of the KasperskyOS kernel
  - Development tools (Clang compiler, LD linker, GDB debugger, QEMU emulator, and accompanying tools)
  - Utilities and scripts (for example, source code generators, `makekss` script for creating the Kaspersky Security Module, and `makeimg` script for creating the solution image)
  - A set of libraries that provide partial compatibility with the POSIX standard
  - Drivers
  - System programs (for example, virtual file system)
  - Usage examples for components of KasperskyOS Community Edition
  - End User License Agreement
  - Information about third-party code (Legal Notices)

- KasperskyOS Community Edition Developer's Guide (Online Help)
- Release Notes

> KasperskyOS Community Edition is delivered as a separate deb package for each of the supported hardware platforms.

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

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

- Image of the KasperskyOS kernel.

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

System requirements

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

1. **Operating system:** Ubuntu GNU/Linux 22.04 (Jammy Jellyfish). A Docker container can be used.
2. **CPU:** processor with an x86-64 architecture.
3. **RAM**: it is recommended to have at least 4 GB of RAM for convenient use of the build tools.
4. **Disk space**: at least 5 GB of free space in the `/opt` folder (depending on the solution being developed).

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

- Raspberry Pi 4 Model B (Revision 1.4, 1.5) with 4 or 8 GB of RAM
- MicroSD card with at least 2 GB
- USB-UART converter

To run examples on the Radxa ROCK 3A hardware platform, the following is required:

- Radxa ROCK 3A (Arm64 architecture) Revision 1.3 with 8 GB of RAM
- MicroSD card with at least 2 GB
- USB-UART converter that supports a speed of 1,500,000 baud

Included third-party libraries and applications

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

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

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

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

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

- **CMake (v.3.25.0)** is a cross-platform software tool that automatically builds software from source code.

  Documentation: https://cmake.org/documentation

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

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

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

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

- **Autoconf (v.2.69)** is a tool that generates `configure` scripts for automatically configuring and building software from source code.

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

- **Libtool (v.2.4.2, v.2.4.7)** is a generic library support script that conceals the complexity of using shared libraries behind a consistent, portable interface.

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

- **Binutils (v.2.41)** is a set of tools for working with binary files that includes an assembler, linker, archiver, and other tools.

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

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

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

- **QEMU (v.8.2.5)** is a program that emulates hardware of various platforms.

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

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

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

- **Boost (v.1.82.0)** is a set of class libraries that utilize C++ language functionality and provide a convenient cross-platform, high-level interface for concise coding of various everyday programming subtasks (such as working with data, algorithms, files, threads, and more).

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

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

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

- **Civetweb (v.1.12)** is an easy-to-use, powerful, embeddable web server based on C/C++ with additional support for CGI, SSL and Lua.

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

- **fmt (v.9.1.0)** is an open-source formatting library.

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

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

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

- **GoogleTest (v.1.10.0)** is a C++ code testing library.

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

- **iperf (v.3.10.1)** is a network performance testing library.

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

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

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

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

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

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

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

- **libpcap (v.1.10.4)** is a library for developing programs that can capture, filter, and analyze network traffic in UNIX-like systems.

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

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

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

- **libxml2 (v.2.10.4)** is a library for working with XML.

  Documentation: http://xmlsoft.org

- **Mbed TLS (v.3.6.1)** is a library that implements cryptographic protocols such as TLS/SSL and DTLS, and algorithms for encryption, hashing, and authentication.

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

- **Eclipse Mosquitto (v2.0.18)** is a message broker that implements the MQTT protocol.

  Documentation: https://mosquitto.org/documentation

- **jsoncpp (v.4.2.8P15)** is a library for working with the NTP time protocol.

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

- **OpenSSL (v.1.1.1t)** is a full-fledged open-source encryption library.

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

- **pcre (v.8.44)** is a library for working with regular expressions.

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

- **spdlog (v.1.11.0)** is a logging library.

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

- **sqlite (v.3.41.2)** is a library for working with databases.

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

- **Zlib (v.1.2.13)** is the data compression library.

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

- **usb (v.14.0.0)** is a library for working with USB devices.

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

- **libevdev (v.1.12.1)** is a library for working with evdev peripheral devices.

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

- **dhcpcd (v.9.4.1)** is a DHCP/DHCPv6 client intended for automatic configuration of network settings on the client side.

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

- **Lwext4 (v.1.0.0)** is a library for working with the ext2/3/4 file systems.

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

- **abseil/abseil-cpp (v.20211102.0)**, a library that extends the standard library of C++ functions.

  Documentation: https://abseil.io/docs/cpp/

- **clang (v.17.0.6)** is a C/C++ compiler.

  Documentation: http://llvm.org/

- **clang-format (v.13.0.1)** is a tool used to automatically format C/C++ code.

  Documentation: http://llvm.org/

- **corrosion-rs/corrosion (v.0.2.2)** is a tool for integrating Rust into an existing CMake project.

  Documentation: https://github.com/corrosion-rs/corrosion

- **google/re2 (v.2022-02-01)** is a library for working with regular expressions.

  Documentation: https://github.com/google/re2

- **mtheall/ftpd (v.2.3.0)** is a File Transfer Protocol (FTP) server.

  Documentation: https://github.com/mtheall/ftpd

- **python (v.3.12.2)** is a general-purpose, high-level programming language.

  Documentation: https://github.com/python/cpython/

- **rust (v.1.59)** is a general-purpose, multi-paradigm compiled programming language.

  Documentation: https://www.rust-lang.org/

- **wpa_supplicant (v.2.10)** is a cross-platform, open implementation of the IEEE 802.11 standard.

  Documentation: https://w1.fi

- **yaml/libyaml (v.0.2.5)** is a library for working with YAML.

  Documentation: https://yaml.org/

See also Information about third-party code.

Limitations and known issues

Because the KasperskyOS Community Edition is intended for educational purposes only, it includes several limitations:

1. When a program is terminated through any method (for example, "return" from the main thread), the resources allocated by the program are not released, and the program goes to sleep. Programs cannot be started repeatedly.

2. You cannot start two or more programs that have the same EDL description.

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

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

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

6. When running examples on the Raspberry Pi 4 Model B hardware platform, the maximum size of the solution image (kos-image file) must not exceed 248 MB, and 146 MB when running on the Radxa ROCK 3A hardware platform.

7. Full functionality of the Rust language cannot be guaranteed.

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

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

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

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

12. Removed support for performance counters from the KasperskyOS kernel in the SDK. Removed the `perfcnt` example from the SDK.

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

14. Only USB 2.0 devices work in USB 3.0 ports of the Radxa ROCK 3A hardware platform.

Critical changes in version 1.3

Due to modifications made to SDK components in version 1.3, you may need to make changes to the application code that was developed using KasperskyOS Community Edition version 1.2 before using that code with KasperskyOS Community Edition version 1.3.

The following critical changes were made to SDK components in version 1.3:

- Removed support for performance counters from the KasperskyOS kernel in the SDK.
- Declarations of the following functions were removed from the SDK: `fork`, `exec*`, `popen*`, and `pclose`. Use of these functions will result in an error during a build.
- Specifying an invalid name of an IPC channel in the init.yaml.in file template will result in an error during a build.
- The toolchain included in the SDK now uses the Clang compiler.
- TLS 1.3 algorithms are now included in the Mbed-TLS component. You must call the `psa_crypto_init()` function before you use the hashing mechanisms for the first time. To ensure correct operation of the Mbed-TLS library, all you have to do is add the `psa_crypto_init()` call before calling any Mbed-TLS function for the first time. This function can be called any number of times. If the first call is successful, all other calls will also be successful.
- Changes to the kdf library:
  - The `KdfGetDeviceFromContainer()` and `KdfEnumContainerNames()` functions have been removed.
  - The `KdfGetDeviceListByTarget()` and `KdfGetDeviceListByTargetSet()` functions now return a container with a handle of the `KdfDevContainerHandle` type.
- The obsolete `SecurityDisconnect` method has been removed from the `Handle.idl` kernel interface.
- The configuration parameter `VFS_BUFFER_SPLIT_SIZE` has been removed. VFS will use `VFS_BUFFER_SIZE` as the upper limit when transmitting data in an IPC arena. The new parameter `VFS_BUFSIZ` is being implemented to configure the size of the I/O buffer (`setbuf`). You will be able to use MDL buffers to read/write large-sized data.
- Support for file access permissions has been added to VFS. When working with files, VFS will now check the file owner bits (`S_IRUSR`, `S_IWUSR`, and `S_IXUSR`) and either allow or deny specific operations. When creating a file and directory, you must verify that all bits are set correctly:
  - The read/write permission bits must be specified for files: `open(file, O_RDWR | O_CREAT, (S_IRUSR | S_IWUSR)`
  - All three bits must be specified for directories `(Read | Write | Execute)`. The `Execute` bit provides the capability to search for files in the directory: `mkdir(dir, S_IRWXU)`

  The `open()` function lets you create files without specifying these bits, therefore you may encounter a situation in which previously created files may stop opening and instead return an `EACCESS` error. You can use the `chmod()` function to change the file permissions.
- In the `Driver.idl` interface, the `GetDeviceEvents()` method has been renamed to `AwaitDeviceEvents()`.
- The initializer function `kl_drivers_Driver *KdfServerInit(KdfServerData *data)` has been replaced with `kl_drivers_Driver *KdfServerInit(void)`.
- The kernel interface `Task::FreeSelfEnv` has become a stub that returns `rcUimplemented`, and the `KnTaskFreeEnv` and `KnTaskGetEnv` functions are no longer thread-safe.
- Writing to `AF_ROUTE` sockets is prohibited. Now, if you attempt to write to the `AF_ROUTE` socket, the `EACCESS` error is returned. To add/delete routes, you must use `ioctl()` and the `ortentry` structure.

- The behavior of the `nk_arena_get()` call has changed. `RTL_NULL` is returned only if there is an error. Otherwise, the correct memory pointer is returned even if zero-sized data is received.

- The values of an IDL type "string" must contain a terminating null byte when passed in IPC messages, even if they are empty strings. Strings composed of zero bytes will no longer be considered valid and will be denied by the Kaspersky Security Module.

- Function prototypes have been changed:

  - `KosString KosCreateStringEx(KosStringRoot *root, const char *str)` was changed to `Retcode KosCreateStringEx(KosStringRoot *root, const char *str, KosString *outStr)`;

  - `KosString KosCreateString(const char *str)` was changed to `Retcode KosCreateString(const char *str, KosString *outStr)`.

- The kernel interface `task.Task` now has a new method named `GetPid`, which is always used when a process is created.

  As a result, the `EntityInit(Ex)` call will start to return an error when there is a strictly configured security policy with a rigid restriction on methods. You must add the new method to the permitted methods in the policy.

  Example:

```
request dst=kl.core.Core {
  match endpoint=task.Task {
    match method=GetPid {
      match src=Einit { grant () }
    }
  }
}
```

- An endpoint of the `kl.drivers.Driver` type has also been added to each SDK-included EDL file containing the `kl.drivers.Block` endpoint.

  For example, the result will look as follows for `ATA.edl`:

```
entity kl.drivers.ATA


security kl.drivers.block.Security


endpoints {
  driver : kl.drivers.Driver
  ata: kl.drivers.Block
}
```

- The set of methods of the `Block.idl` endpoint has been refined:

  - The `Fini()` method has been removed.

  - The `EnumPorts()` method has been removed. You should use the `GetDeviceList()` method of the `kl.drivers.Driver` endpoint.

  - The `Open()` method has been removed. You should use the `OpenDevice()` method of the `kl.drivers.Driver` endpoint.

  - The `Close()` method has been removed. You should use the `CloseDevice()` method of the `kl.drivers.Driver` endpoint.

- A list of supported codes (MIB) of the `sysctl()` function has been added. A call with codes that are different from the supported codes is prohibited and returns the `ENOSYS` code. All authorized codes have been converted into separate interface methods of the VFS component (`VfsNetConfig.idl`). With security policies, you can permit read-only or write-only by using the `valOperation` argument of an IPC request (except `IpctlForwarding`, `RtDump`, and `RtIflist`): 0 is for writing, or setting a parameter value, 1 is for reading a parameter, and 2 is for requesting the parameter size)

  The supported codes are listed in the table below.

  Authorized codes of the sysctl() function

Parameter name	MIB code	VFS interface method
`net.inet.ip.forwarding`	`CTL_NET, PF_INET, IPPROTO_IP, IPCTL_FORWARDING`	`IpctlForwarding`
`net.inet.ip.mtudisc`	`CTL_NET, PF_INET, IPPROTO_IP, IPCTL_MTUDISC`	`IpctlMtudisc`
`net.inet.ip.ttl`	`CTL_NET, PF_INET, IPPROTO_IP, IPCTL_DEFTTL`	`IpctlTtl`
`net.inet.tcp.keepcnt`	`CTL_NET, PF_INET, IPPROTO_TCP, TCPCTL_KEEPCNT`	`TcpctlKeepcnt`
`net.inet.tcp.keepidle`	`CTL_NET, PF_INET, IPPROTO_TCP, TCPCTL_KEEPIDLE`	`TcpctlKeepidle`
`net.inet.tcp.keepintvl`	`CTL_NET, PF_INET, IPPROTO_TCP, TCPCTL_KEEPINTVL`	`TcpctlKeepintvl`
`net.inet.tcp.mss_ifmtu`	`CTL_NET, PF_INET, IPPROTO_TCP, TCPCTL_MSS_IFMTU`	`TcpctlMssifmtu`
`net.inet.tcp.mssdflt`	`CTL_NET, PF_INET, IPPROTO_TCP, TCPCTL_MSSDFLT`	`TcpctlMssdflt`
`net.inet.tcp.recvspace`	`CTL_NET, PF_INET, IPPROTO_TCP, TCPCTL_RECVSPACE`	`TcpctlRecvspace`
`net.inet.tcp.sendspace`	`CTL_NET, PF_INET, IPPROTO_TCP, TCPCTL_SENDSPACE`	`TcpctlSendspace`
`net.inet.udp.recvspace`	`CTL_NET, PF_INET, IPPROTO_UDP, UDPCTL_RECVSPACE`	`UdpctlRecvspace`
`net.inet.udp.sendspace`	`CTL_NET, PF_INET, IPPROTO_UDP, UDPCTL_SENDSPACE`	`UdpctlSendspace`
`net.route.rtdump`	`CTL_NET, PF_ROUTE, NET_RT_DUMP`	`RtDump`
`net.route.rtiflist`	`CTL_NET, PF_ROUTE, NET_RT_IFLIST`	`RtIflist`
`net.inet.ip.dad_count`	`CTL_NET, PF_INET, IPPROTO_IP, IPCTL_DAD_COUNT`	`IpctlDadcount`
`kern.hostname`	`CTL_KERN, KERN_HOSTNAME`	`KernHostname`

Overview of KasperskyOS

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

See also:

- What's new
- About KasperskyOS Community Edition
- System requirements
- Getting started

Overview

Microkernel

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

Processes and endpoints

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

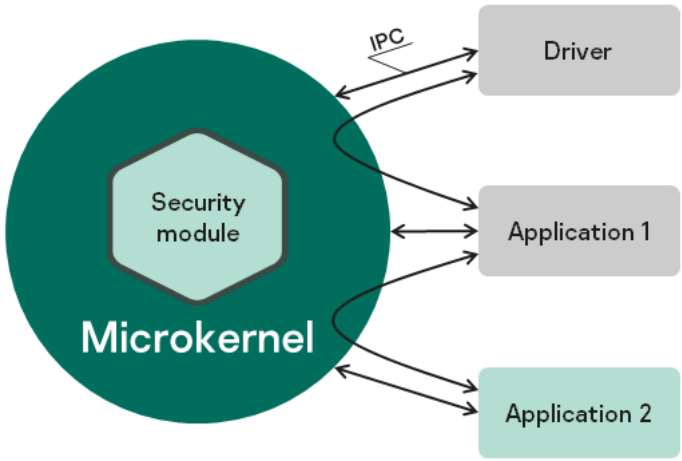
- It can provide endpoints to other processes and/or use the endpoints of other processes via the IPC mechanism.
- It uses core endpoints via the IPC mechanism.
- It is associated with security rules that regulate the interactions of the process with other processes and with the kernel.

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

Implementation of the MILS and FLASK architectural approaches

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

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



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

KasperskyOS-based solution

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

Security policy for a KasperskyOS-based solution

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

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

Kaspersky Security System technology

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

Source code generators

Some of the source code of a KasperskyOS-based solution is created by source code generators. Specialized programs generate the source code in C from declarative descriptions. They generate source code of the Kaspersky Security Module, source code of the *initializing program* (which starts all other programs in the solution and statically defines the topology of interaction between them), and the source code of the methods and types for carrying out IPC (*transport code*).

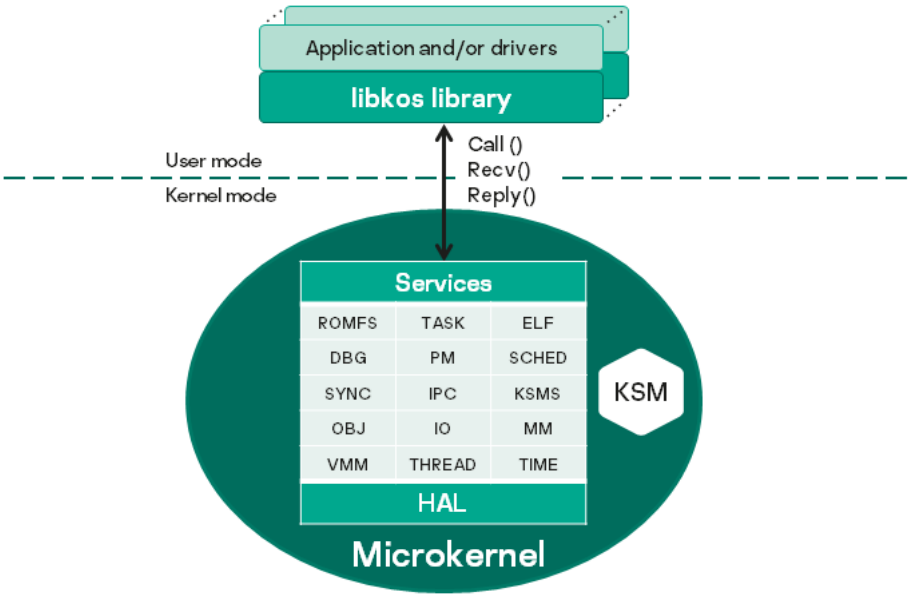
The transport code is generated by the `nk-gen-c` compiler based on the formal specifications of solution components.

The source code of the Kaspersky Security Module is generated by the `nk-psl-gen-c` compiler from the solution security policy description and formal specifications of solution components.

The source code of the initializing program is generated by the `einit` tool from the init description and formal specifications of solution components.

KasperskyOS architecture

The KasperskyOS architecture is presented in the figure below:



KasperskyOS architecture

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

Kernel subsystems and their purpose

Designation	Name	Purpose
HAL	Hardware abstraction subsystem	Basic hardware support: timers, interrupt controllers, memory management unit (MMU). This subsystem includes UART drivers and low-level means for power management.
IO	I/O manager	Registration and deallocation of hardware platform resources required for the operation of drivers, such as Interrupt ReQuest (IRQ), Memory-Mapped Input-Output (MMIO), I/O ports, and DMA buffers. If the hardware platform has an input–output memory management unit (IOMMU), this subsystem guarantees the allocation of memory used by devices.

MM	Physical memory manager	Allocation and deallocation of physical memory pages, distribution of physically contiguous page areas.
VMM	Virtual memory manager	Management of physical and virtual memory: reserving, committing, and releasing memory. Working with memory page tables for insulating the address spaces of processes.
THREAD	Thread manager	Management of threads: creating, terminating, locking, and resuming threads.
TIME	Real-time clock subsystem	Getting the time and setting the system clock. Using clocks provided by hardware.
SCHED	Scheduler	Scheduling of threads: standard threads, real-time threads, and idle threads.
SYNC	Synchronization primitive support subsystem	Implementation of basic synchronization primitives: spinlocks, mutexes, and events. The kernel supports only one primitive—futex. All other primitives are implemented based on a futex in the user space.
IPC	Interprocess communication subsystem	Implementation of a synchronous IPC mechanism based on the rendezvous principle.
KSMS	Security module interaction subsystem	This subsystem is used for working with the security module. It provides all messages relayed via IPC to the security module so that these messages can be checked.
OBJ	Object manager	Management of the general behavior of all KasperskyOS resources: tracking their life cycle and assigning unique security IDs (for details, see Resource Access Control). This subsystem is closely linked to the capability-based access control mechanism (OCap).
ROMFS	Immutable file system image startup subsystem	Operations with files from ROMFS: opening and closing, receiving a list of files and their descriptions, and receiving file characteristics (name, size).
TASK	Process management subsystem	Management of processes: creating, starting, and terminating processes. Receiving information about running processes (such as names and paths) and their exit codes.
ELF	Executable file loading subsystem	Loading executable ELF files from ROMFS into RAM, parsing headers of ELF files.
DBG	Debug support subsystem	Debugging mechanism based on GDB (GNU Debugger). The availability of this subsystem in the kernel is optional.
PM	Power manager	Power management: restart and shutdown.

IPC

IPC mechanism

Exchanging IPC messages

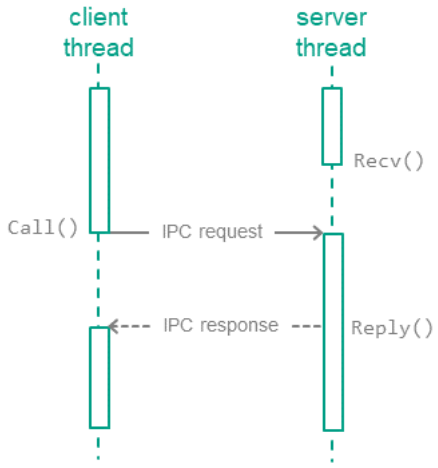
In KasperskyOS, processes interact with each other by exchanging IPC messages (*IPC request* and *IPC response*). In an interaction between processes, there are two separate roles: *client* (the process that initiates the interaction) and *server* (the process that handles the request). Additionally,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

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

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

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

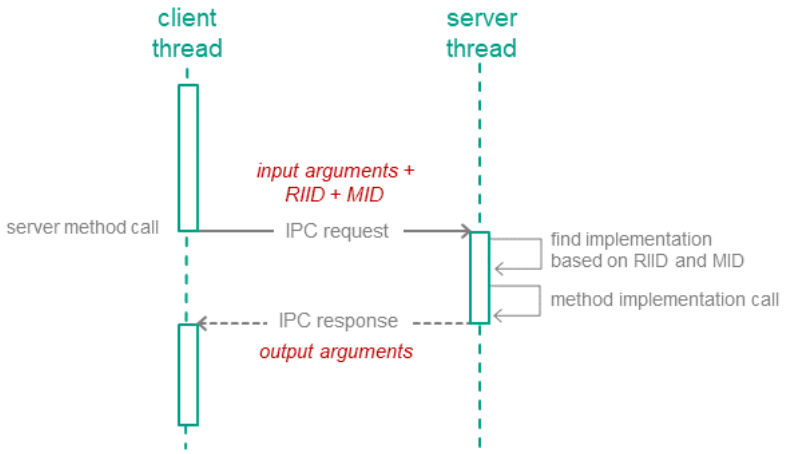
3. When an IPC response is received, the client thread is unlocked and continues execution.



Exchanging IPC messages between a client and a server

Calling methods of server endpoints

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



Calling a server endpoint method

IPC channels

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

KasperskyOS has two mechanisms for creating IPC channels:

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

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

IPC control

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

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

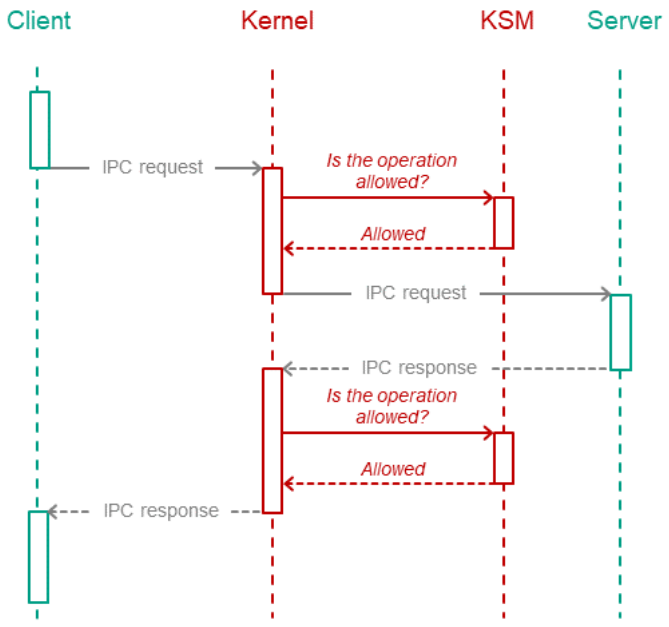
1. The security module verifies that the IPC message complies with the called method of the endpoint (the size of the IPC message is verified along with the size and location of certain structural elements).

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

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

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

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



Controlled exchange of IPC messages between a client and a server

Transport code for IPC

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

Transport code for developed components of a solution

A developer of a KasperskyOS-based solution component can generate transport code based on IDL, CDL and EDL descriptions related to this component. The KasperskyOS SDK includes the `nk-gen-c` compiler for this purpose. The `nk-gen-c` compiler generates transport methods and types for use by both a client and a server.

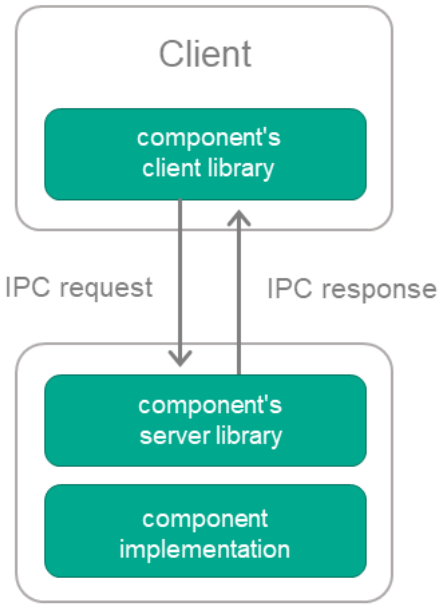
Transport code for supplied components of a solution

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

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

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

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



Using a supplied solution component via IPC

IPC between a process and the kernel

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

A solution developer is not required to create IPC channels between processes and the kernel because these channels are created automatically when processes are created. (To set up interaction between processes, the solution developer has to create IPC channels between them.)

The Kaspersky Security Module makes decisions regarding interaction between processes and the kernel the same way it makes decisions regarding interaction between a process and other processes. (The KasperskyOS SDK has IDL, CDL and EDL descriptions for the kernel that are used to generate source code of the security module.)

Resource Access Control

Types of resources

KasperskyOS has two types of resources:

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

Handles

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

Security identifiers

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

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

Security context

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

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

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

Resource access control by the KasperskyOS kernel

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

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

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

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

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

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

Resource access control by resource providers

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

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

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

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

Structure and startup of a KasperskyOS-based solution

Structure of a solution

The image of the KasperskyOS-based solution loaded into hardware contains the following files:

- Image of the KasperskyOS kernel
- File containing the executable code of the Kaspersky Security Module
- Executable file of the initializing program
- Executable files of all other solution components (for example, applications and drivers)
- Files used by programs (for example, dynamic libraries, files containing settings, fonts, graphical and audio data)

The ROMFS file system is used to save files in the solution image.

Starting a solution

A KasperskyOS-based solution is started as follows:

1. The bootloader starts the KasperskyOS kernel.
2. The kernel finds and loads the security module (as a kernel module).
3. The kernel starts the initializing program.
4. The initializing program starts the programs included in the solution (one, several, or all).

The KasperskyOS Developer's Quick Start Guide

Preparations for developing

To start developing KasperskyOS solutions, complete the following steps:

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

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

2. Carefully read the Overview of KasperskyOS.

3. Carefully read this section.

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

Exploring the examples from KasperskyOS Community Edition

The `examples` directory in KasperskyOS Community Edition contains examples of KasperskyOS solution projects with a `README.md` description and comments in text files. Each example resides in a separate directory. To build an example, copy it into a directory that you have write permissions to, such as home, and run `cross-build.sh`. The build process will create a solution image: `<example directory>/build/einit/kos-*image`. If `cross-build.sh` calls `cmake` with the `--target sim` parameter, the built image will be automatically run in QEMU. You can also build examples to be run on Raspberry Pi 4 B or Radxa ROCK 3A. For more details, refer to the Building and running examples section.

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

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

Location of libraries and executable files of supplied solution components

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

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

Creating a KasperskyOS-based solution project.

You are advised to use the CMake build system, but you may also use another build system or create your own build script by using the build tools from KasperskyOS Community Edition.

To create a CMake project for a KasperskyOS solution, create a hierarchy of directories and files as described in Using CMake from the contents of KasperskyOS Community Edition, similarly to the project examples. In addition to files containing source code, a solution CMake project includes an `*.sh` build script, `CMakeLists.txt` files, an `init.yaml(.in)` file, `security.psl.in` and `*.psl` files, and `*.idl`, `*.cdl` and `*.edl` files.

Creating a build script (*.sh file)

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

Creating build scenarios (CMakeLists.txt files)

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

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

Creating an init description (init.yaml.in and init.yaml files)

During the build process, `init.yaml.in` is used to create an `init.yaml` file, which is an init description. The macros specified in `init.yaml.in` are expanded in the `init.yaml` file. The init description specifies the processes, except for the process of the initializing program, and the IPC channels that will be created when the solution starts. It can also specify the run parameters and environment variables of programs. For example, you can use macros in `init.yaml.in` to populate `init.yaml` with IPC channels, run parameters, and environment variables of programs defined via the `set_target_properties()` CMake command in `CMakeLists.txt` files (see CMakeLists.txt files for building applications). In addition, `init.yaml.in` supports the ability to specify variables defined by the CMake command `set()` in `CMakeLists.txt` files.

To create the `init.yaml.in` file, use the syntax described in Overview: Einit and init.yaml, and you can use macros.

You can also use the system program ExecutionManager to start processes.

Creating a solution security policy description (security.psl.in and *.psl files)

During the build process, `security.psl.in` is used to create a `security.psl` file, which is the top-level file of the solution security policy description. The macros specified in `security.psl.in` are expanded in `security.psl` in the form of declarations to include `*.psl` files located in `sysroot-*-kos/include/kl`, in KasperskyOS Community Edition. The `security.psl.in` file also supports the ability to specify variables defined by the `set()` CMake command in `CMakeLists.txt` files.

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

To create `security.psl.in` and other `*.psl` files, such as `core.psl`, carefully read Describing a security policy for a KasperskyOS solution and Security.psl.in template. You need a complete understanding of what must be allowed and denied for each program in the solution and under which conditions.

The `security.psl.in` file and other `*.psl` files can include declarations to perform a security audit and test the policy.

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

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

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

The `*.edl` file is mandatory, while `*.cdl` and `*.idl` are optional. If the program does not provide endpoints to other programs in the solution, its formal specification may consist of one `*.edl` file. For example, the formal specification of the initializing program consists of one `Einit.edl` file created automatically during the build.

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

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

Creating initializing program

KasperskyOS solutions include an initializing program. Each solution project contains an `einit` directory. The name of this directory matches the name of the initializing program but does not contain its source code because the code is generated automatically during the build.

Creating IPC channels

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

Creating and using transport code

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

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

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

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

Querying the Kaspersky Security Module

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

Working with IPC messages

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

Using dynamic libraries

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

Working with file systems and network stack

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

See File systems and network.

Working with KPA packages

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

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

- Tools for managing KPA packages let you build a KPA package from program source files in the system where the KasperskyOS Community Edition is installed and then install the KPA package in a built KasperskyOS-based solution image.

- PackageManager component lets you install KPA packages in an operational KasperskyOS-based solution, remove KPA packages, and receive information about them.

See Working with KPA packages.

KasperskyOS APIs

KasperskyOS provides the following APIs:

- POSIX (with limitations and specific features of implementation) and other APIs of the standard C library:

- Main APIs (header files in the `sysroot-*-kos/include/strict` and `sysroot-*-kos/include/sys` directories from KasperskyOS Community Edition)
- Extended APIs (header files in `sysroot-*-kos/include` from KasperskyOS Community Edition)

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

- Native APIs:
  - Top-level APIs (header files in `sysroot-*-kos/include/kos`, in KasperskyOS Community Edition)
  - Low-level APIs (header files in `sysroot-*-kos/include/coresrv`, in KasperskyOS Community Edition)

Components for application development

KasperskyOS Community Edition includes the following:

- The MessageBus component that implements the message bus.
- The LogRR component, which provides a system for logging information about the operation of other programs.

Debugging programs

To debug programs, use the GDB debugger from KasperskyOS Community Edition. Debugging can be performed in QEMU or on the hardware platform. To perform debugging in QEMU, you should use the GDB server of QEMU or the GDB server of the KasperskyOS kernel. To perform debugging on the hardware platform, you must use the GDB server of the KasperskyOS kernel.

See Debugging programs in a KasperskyOS-based solution.

Developing drivers

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

See Developing drivers for KasperskyOS.

Using a glossary

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

Getting started

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

Using a Docker container

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

To use a Docker container for installing KasperskyOS Community Edition:

1. Make sure that the Docker software is installed and running.
2. To download the official Docker image of the Ubuntu GNU/Linux 22.04 (Jammy Jellyfish) operating system from the public Docker Hub repository, run the following command:

   `docker pull ubuntu:22.04`
3. To run the image, run the following command:

   `docker run --net=host --user root --privileged -it --rm ubuntu:22.04 bash`
4. Copy the deb package for installation of KasperskyOS Community Edition into the container.
5. Install KasperskyOS Community Edition.

Installation and removal

Installation

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

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

```
$ sudo apt update && sudo apt install <path-to-deb-package>
```

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

To conveniently work with tools provided in the KasperskyOS Community Edition SDK, the path to the executable files of these tools in `/opt/KasperskyOS-Community-Edition-<version>/toolchain/bin` must be added to the `PATH` environment variable. To avoid having to do this each time you log in to a user session, run the script `/opt/KasperskyOS-Community-Edition-<version>/common/set_env.sh`, log out and log in to a session again.

Syntax of the command for calling the `set_env.sh` script:

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

Parameters:

- `-d`

  Cancels the action of the script.

- `-h`, `--help`

  Displays the Help text.

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

Removal

Prior to removing KasperskyOS Community Edition, cancel the action of the `set_env.sh` script if you ran this script after installing the SDK.

To remove KasperskyOS Community Edition, run the following command:

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

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

Configuring the development environment

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

Configuring the code editor

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

- Install code editor extensions and plugins for your programming language (C, C++ and/or Rust).

- Add the header files included in KasperskyOS Community Edition to the development project.

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

Example of how to configure Visual Studio Code

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

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

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

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

   A workspace can be opened implicitly by using the `File` > `Open folder` menu options.

2. Make sure the C/C++ for Visual Studio Code extension is installed.

3. In the `View` menu, select the `Command Palette` item.

4. Select the `C/C++: Edit Configurations (UI)` item.

5. In the `Include path` field, enter `/opt/KasperskyOS-Community-Edition-<version>/sysroot-*-kos/include`.

6. Close the `C/C++ Configurations` window.

KasperskyOS SDK Extension for Visual Studio Code

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

Installing and removing the extension

Installing dependencies

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

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

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

2. Find and download the relevant extensions:

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

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

   - `devcontainers` is required for Docker support.

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

4. Open the Visual Studio Code development environment.

5. Press **F1**.

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

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

Installing the extension

To install KasperskyOS SDK Extension, do the following:

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

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

Removing the extension

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

Extension functions

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

After startup, the extension adds the following buttons to the lower panel of the Visual Studio Code editor:

- all – select the build target.
- – start the project in QEMU.
- – start the selected target with debugging.
- – build all targets.
- – build the selected target.
- – clear the build directory.
- type=debug – switch the build type.
- With tests – enable/disable the project build with tests.
- aarch64-kos – select the target platform for the build.
- No device – select the device or QEMU emulator.

- CE-RPi4-1.3.0 – display the SDK version.

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

Working with a basic KasperskyOS image

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

1. Running a basic KasperskyOS-based solution image, included in the SDK, in the QEMU emulator. The basic image contains all the system programs required to run and debug programs packaged in KPA packages.

2. Build an application and package it into a KPA package. You must use the `CMake` commands of the kpa library to pack a program into a KPA package.

3. Install a program from a KPA package into the solution image that was run at step 1 and start it under KasperskyOS.

4. Make changes to the application code.

5. Repeat steps 2–5.

6. With the extension, you can also debug a program that is running in a basic KasperskyOS image. For more details, refer to Debugging programs that are part of a KPA package.

Create a basic solution security policy description

The extension lets you automatically create basic solution security policy descriptions for a project.

To create a basic security policy description:

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

2. Press **F1**.

3. Run the command `KOS: Generate policy file for current project`.

4. Select the type of basic policy description that you want to create:

   - `Grant all permissions`: create a basic solution security policy description that allows all interactions between different processes of any class and between these processes and the KasperskyOS kernel, and that also allows any process to initialize the startup of processes.

   - `Grant necessary permissions`: create a basic solution security policy description that allows all interactions between different processes of any class and between these processes and the KasperskyOS kernel, and that also determines the necessary permissions for initializing the startup of processes based on the init.yaml.in file within the project.

5. Enter the file name for saving the created security policy description.

> These policies can be used only as stubs during the early stages of development of a KasperskyOS-based solution so that the Kaspersky Security Module does not interfere with interactions. It would be unacceptable to apply such policies in a real-world KasperskyOS-based solution.

Running tests

After the extension is activated, the Testing tab ( ) appears on the side panel of Visual Studio Code. When you select the Testing tab, the code editor window displays a tree of tests created using the Google Test library and found in files in the `test/` directory.

To run the tests, tap the button. All detected tests are run by default. However, you can select specific tests on the tabs containing the source code of the tests, or you can select specific groups of tests by selecting them from the tree of all tests. You can only run tests that are added to `CMakeLists.txt` files for building programs using the `kl_kos_add_small_test()` or `generate_kos_test()` `CMake` commands from the TestGenerator `CMake` library included in the SDK.

The test results and log files are located in the directory `<build_directory>/bin/tests/output`.

Starting solution security policy tests

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

To build and run solution security policy tests:

1. Make sure that the solution security policy description contains the solution security policy tests.
2. Make sure that the `CMake` commands for building solution security policy tests have been added to one of the `CMakeLists.txt` files of the project.
3. Make sure that the KasperskyOS SDK Extension is installed and active.
4. Build all targets by clicking the button in the lower panel.
5. Click the build target selection button all and select the target named `kos-qemu-image-PalTest<N>-sim` (N refers to the PSL file index within the list of PSL files containing solution security policy tests).
6. Click the button to build the selected target.

Starting a basic KasperskyOS-based solution image

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

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

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

2. Click the device selection button No device on the lower panel or press `F1` and run the command `KOS: Select Device`.

3. Select **Create new emulator**.

4. Select **kos_base-dev**.

5. In the opened field, enter the additional QEMU flags and press **`Enter`**, or immediately press **`Enter`** to use the default flags.

6. In the opened field, enter the name of the new emulator and press **`Enter`**, or immediately press **`Enter`** to use the automatically generated name.

7. Wait until the previous command completes.

8. In the notification about the new emulator being created, click the **Start** button.

9. Wait for the emulator to finish loading.

Building programs in Visual Studio Code

To build a program in Visual Studio Code:

1. Open the program project directory in Visual Studio Code.

2. Make sure that the project was correctly identified based on the appearance of additional extension buttons in the lower panel. If the buttons do not appear, manually activate the solution by running the command **KOS: Activate extension in this directory**.

3. Select the build architecture by clicking the button for selecting the target platform for the build aarch64-kos.

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

Starting programs in a basic solution image

If a program that you want to start uses file systems (via the VFS component), the following is required before starting the program:

a. Use the menu to open the extension settings window: **File** → **Preferences** → **Settings**, then select **Extensions** → **KasperskyOS**.

b. Set the `client:kl.VfsSdCardFs` value for the `VFS_FILESYSTEM_BACKEND` environment variable in the `Application Environment Variables` parameter.

To start a program in a basic solution image:

1. Open the program project directory in Visual Studio Code.

2. Make sure that the project was correctly identified based on the appearance of additional extension buttons in the lower panel. If the buttons do not appear, manually activate the solution by running the command **KOS: Activate extension in this directory**.

3. Make sure that the basic solution image is started according to the instructions provided in the Starting a basic KasperskyOS-based solution image section.

4. Click the button for selecting the device or QEMU emulator No device in the lower panel.

5. Select the previously started basic solution image.

6. Build the program according to the instructions provided in the Building a program in Visual Studio Code section.

7. Make sure that the program has been packed into a KPA package.

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

8. Click the target selection button all. In the drop-down list, select the built program KPA package that is signed as `[application]` in the list.

9. Click the start button ▷. The program will be installed to the selected basic image and will automatically start.

Debugging a program that is part of a KPA package

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

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

1. Install the KasperskyOS SDK Extension.

2. Build an application and package it into a KPA package. You must use the `CMake` commands of the kpa library to pack a program into a KPA package.

3. Run a basic KasperskyOS image from KasperskyOS Community Edition with a GDB server built into the kernel.

4. Connect the debugger to the GDB server of the kernel. The debugger can be connected at program startup or while the program is running. Details:

   - Connecting a debugger at program startup.

   - Connecting a debugger while a program is running.

5. Debugging a program using the Visual Studio Code graphical user interface. For more details, refer to the Visual Studio Code documentation: https://code.visualstudio.com/docs/editor/debugging .

Connecting a debugger when starting a program in a basic image

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

1. Set a breakpoint in the program source code before the fragment to be debugged.

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

3. Run a basic KasperskyOS image from KasperskyOS Community Edition.

4. Wait for the basic image to finish loading and install the program by running the command `KOS: Install package`.

5. On the side bar of Visual Studio Code, click **Run and debug** > **create a launch.json file** and then select **KasperskyOS Debugger**.

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

6. In the `launch.json` file, in the field of the configuration named `(kos/gdb) Launch & debug application`, specify the path to the binary file of your program resulting from the build, and the program name. In the `eiid` field, you need to specify the `kl.Kds` value.

7. Start debugging by clicking the `(kos/gdb) Launch & debug application` button on the lower pane.

8. In the drop-down list, select the configuration named `(kos/gdb) Launch & debug application`.

9. The debug console will show a message stating that the program is ready for debugging. Click **Continue**, and program execution will be stopped at the breakpoint that you selected at step 1.

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

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

1. Add an infinite loop to the `main` function of the program. This enables the debugger to connect to the running program.

2. Set a breakpoint in the program source code before the fragment to be debugged.

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

4. Run a basic KasperskyOS image from KasperskyOS Community Edition.

5. On the side bar of Visual Studio Code, click **Run and debug** > **create a launch.json file** and then select **KasperskyOS Debugger**.

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

6. In the `launch.json` file, in the field of the configuration named `(kos/gdb) Attach to process`, specify the path to the binary file of your program resulting from the build.

7. Start the program in the basic image.

8. Start debugging by clicking the `(kos/gdb) Attach to process` button on the lower pane.

9. In the drop-down list, select the configuration named `(kos/gdb) Attach to process` and select the name of the program to debug.

10. The debug console will show a message stating that the program is ready for debugging. Click **Continue**, and program execution will be stopped at the breakpoint that you selected at step 2.

Extension settings

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

Extension settings

`Application Arguments`

List of command-line arguments for applications started with the extension.

`Application Environment Variables`

List of environment variables for applications started with the extension.

`Auto_update`

Enable/disable automatic updates of the extension when the `SDK Path` parameter is modified.

`Build Threads Num`

Select the quantity of processor threads used during the build. All available threads are used by default.

`Cmake Build Flags`

Additional CMake flags for the build.

`Cmake Config Flags`

Additional configuration flags for running CMake.

`Auto Open Log Tab`

Enable/disable automatic opening of the tab containing the command execution log.

`Force Clean`

The Clean button deletes the build directory.

`Gdbinit File`

Path to the `gdbinit` file containing the GDB commands that are executed when the debugger is started.

`Run No Graphic`

Enable/disable enforced startup of the emulator without a graphical user interface.

`SDK Path`

Path to the installed KasperskyOS Community Edition.

If the extension does not detect the correct SDK in the `SDK Path` at startup, you will be prompted to select an SDK from the list of detected SDKs. To avoid having to do this for each workspace, it is recommended to set the `SDK Path` for the User settings.

`Set Build Dir`

Project build directory. The `./build` directory is used by default.

`Shell Export Variables`

Additional variables that are set before starting `CMake`.

`Suppress Extensions Recommendations`

Enable/disable the reminder that you need to set the dependencies of the KasperskyOS SDK Extension.

`Test Failures Num`

Number of failed tests displayed in the log.

`Test Failures Only`

Enable/disable display of only failed tests in the log.

`Test Files Pattern`

Mask for searching for files containing the source code of project tests.

`Test No Skipped`

Enable/disable display of skipped tests in the log.

`Test Skip Details`

Enable/disable display of skipped and failed tests in the log.

`Test Threads Num`

Select the quantity of processor threads used when running tests. All available threads are used by default. However, single tests will use one thread each.

`Test Timeout`

Time limit for running each test (in seconds). When this value is set to `0`, the limits from the test configurations are applied.

`Test Verbose`

Enable/disable display of debug information about tests in the log.

`Trace: Server`

Select the level of logging detail for the Visual Studio Code LSP client when working with the LSP server built into the SDK for the IDL/CDL/EDL languages. You can view the LSP client log by selecting the `nk-lsp` channel in the Visual Studio Code output window.

`User SDKs`

Additional paths to SDKs for displaying the SDK selection menu

Building and running examples

Building the examples

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

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

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

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

Building the examples to run on QEMU

To build an example, go to the directory with the example and run this command:

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

Running the `cross-build.sh` script creates a KasperskyOS-based solution image that includes the example, and initiates startup of the example in QEMU. The `kos-qemu-image` solution image is located in the `<name of example>/build/einit` directory.

Building the examples to run on Raspberry Pi 4 B or Radxa ROCK 3A

To build an example, go to the directory with the example and run this command:

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

The type of image that is created by the `cross-build.sh` script depends on the value that you choose for the `target` parameter:

- `kos-image`

  This creates a KasperskyOS-based solution image that includes the example. The `kos-image` solution image is located in the `<name of example>/build/einit` directory.

- `sd-image`

  This creates a file system image for a bootable SD card. The following is loaded into the file system image: `kos-image`, U-Boot bootloader that starts the example, and the firmware for Raspberry Pi 4 B or Radxa ROCK 3A. The source code for the U-Boot bootloader and firmware can be downloaded from the website https://github.com. The file system image file `hdd.img` is saved in the directory `<example name>/build`.

> Before running the examples on Radxa ROCK 3A, you also need to build the drivers provided as a source code in the SDK. Instructions for building drivers can be found in the descriptions of examples (README.md files).

Running examples on QEMU

Running examples on QEMU on Linux with a graphical shell

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

An example is run on QEMU on Linux with a graphical shell using the `cross-build.sh` script, which also builds the example. To run the script, go to the folder with the example and run the command:

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

Running examples on QEMU on Linux without a graphical shell

To run an example on QEMU on Linux without a graphical shell, go to the directory with the example, build the example and run the following commands:

```
$ cd build/einit
# Before running the following command, be sure that the path to
# the directory with the qemu-system-aarch64 executable file is saved in
# the PATH environment variable. If it is not there,
# add it to the PATH variable.
$ 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
```

Preparing Raspberry Pi 4 B to run examples

Connecting a computer and Raspberry Pi 4 B

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

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

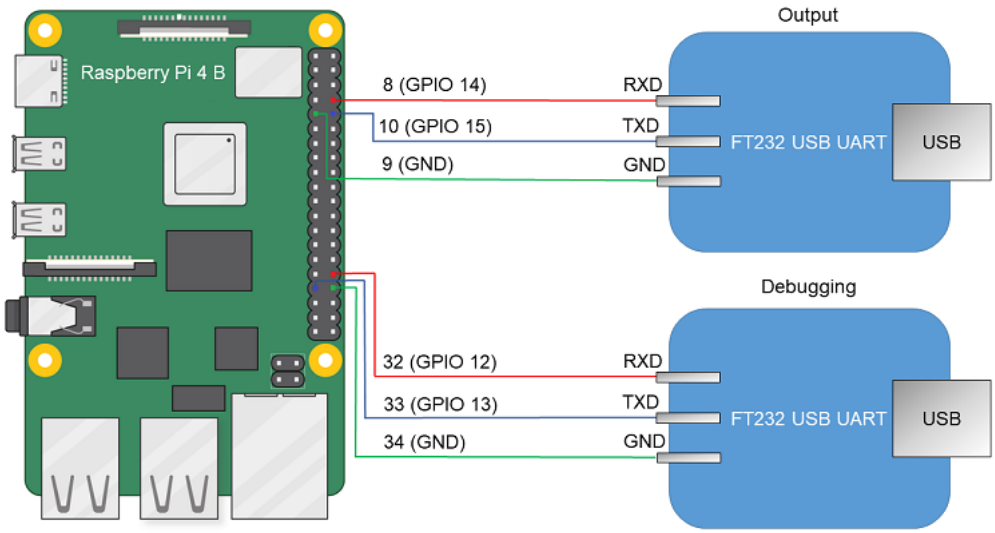


Diagram for connecting USB-UART converters and Raspberry Pi 4 B

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

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

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

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

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

Preparing a bootable SD card for Raspberry Pi 4 B

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

```
# In the following command, path_to_img is the path to the image file
# [X] is the final character in the name of the SD card block device.
$ sudo pv -L 32M path_to_img | sudo dd bs=64k of=/dev/sd[X] conv=fsync
```

If `kos-image` was created when building the example, the SD card requires additional preparations before you can write the image to it. A bootable SD card for Raspberry Pi 4 B can be prepared automatically or manually. After preparing the SD card, you must copy the `kos-image` file from the directory `<example name>/build/einit` to the bootable area (FAT32 partition) of the prepared SD card.

To automatically prepare the bootable SD card, connect the SD card to the computer and run the following commands:

```
# To create a bootable drive image file (*.img),
# run the script:
$ sudo /opt/KasperskyOS-Community-Edition-<version>/common/rpi4_prepare_sdcard_image.sh
# In the following command, path_to_img is the path to the image file
# of the bootable drive (this path is displayed upon completion
# of the previous command), [X] is the final character
# in the name of the SD card block device.
$ sudo pv -L 32M path_to_img | sudo dd bs=64k of=/dev/sd[X] conv=fsync
```

To manually prepare the bootable SD card:

1. If a Docker container is used when working with KasperskyOS Community Edition, it must be granted access to devices in the `/dev` directory on the host system. To do so when starting the Docker container, add the following parameter to the startup command:

   ```
   -v /dev:/dev
   ```

2. Build the U-Boot bootloader for ARMv8, which will automatically run the example. To do this, run the following commands:

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

3. Prepare the image containing the file system for the SD card.

   ```
   # Image will contain a boot partition of 1 GB in fat32 and three partitions of 350 MB each in ext2, ext3 and
   ext4, respectively.
   $ fs_image_name=sdcard.img
   $ dd if=/dev/zero of=${fs_image_name} bs=1024k count=2048
   $ sudo parted ${fs_image_name} mklabel msdos
   $ loop_device=$(sudo losetup --find --show --partscan ${fs_image_name})
   $ sudo parted ${loop_device} mkpart primary fat32 8192s 50%
   $ sudo parted ${loop_device} mkpart extended 50% 100%
   $ sudo parted ${loop_device} mkpart logical ext2 50% 67%
   $ sudo parted ${loop_device} mkpart logical ext3 67% 84%
   ```

```
$ sudo parted ${loop_device} mkpart logical ext4 84% 100%
$ sudo parted ${loop_device} set 1 boot on
$ sudo mkfs.vfat ${loop_device}p1
$ sudo mkfs.ext2 ${loop_device}p5
$ sudo mkfs.ext3 ${loop_device}p6
$ sudo mkfs.ext4 -O ^64bit,^extent ${loop_device}p7
```

4. Copy the U-Boot bootloader and embedded software (firmware) for Raspberry Pi 4 B to the received file system image by running the following commands:

```
# In the following commands, the path ~/mnt/fat32 is just an example.
# You can use a different path.
$ mount_temp_dir=~/mnt/fat32
$ mkdir -p ${mount_temp_dir}
$ sudo mount ${loop_device}p1 ${mount_temp_dir}
$ git clone --depth 1 --branch 1.20220331 https://github.com/raspberrypi/firmware.git firmware
$ sudo cp u-boot.bin ${mount_temp_dir}/u-boot.bin
$ sudo cp -r firmware/boot/. ${mount_temp_dir}
```

5. Fill in the configuration file for the U-Boot bootloader in the image by using the following commands:

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

6. Write the resulting image to an SD card. To do this, connect the SD card to the computer and run the following command:

```
# In the following command, [X] is the last symbol in the name of the block device for the SD card.
$ sudo pv -L 32M ${fs_image_name} | sudo dd bs=64k of=/dev/sd[X] conv=fsync
```

7. After preparing the SD card, you must copy the `kos-image` file from the directory `<example name>/build/einit` to the bootable area (FAT32 partition) of the prepared SD card.

Preparing Radxa ROCK 3A to run examples

Switching for a computer and Radxa ROCK 3A

To see the output from Radxa ROCK 3A on a computer and to have debug capabilities, do the following:

1. Connect the pins of the USB-UART converters to the corresponding GPIO pins of the Radxa ROCK 3A (see the figure below). If debugging is not necessary, all you need to do is connect one USB-UART converter for output.

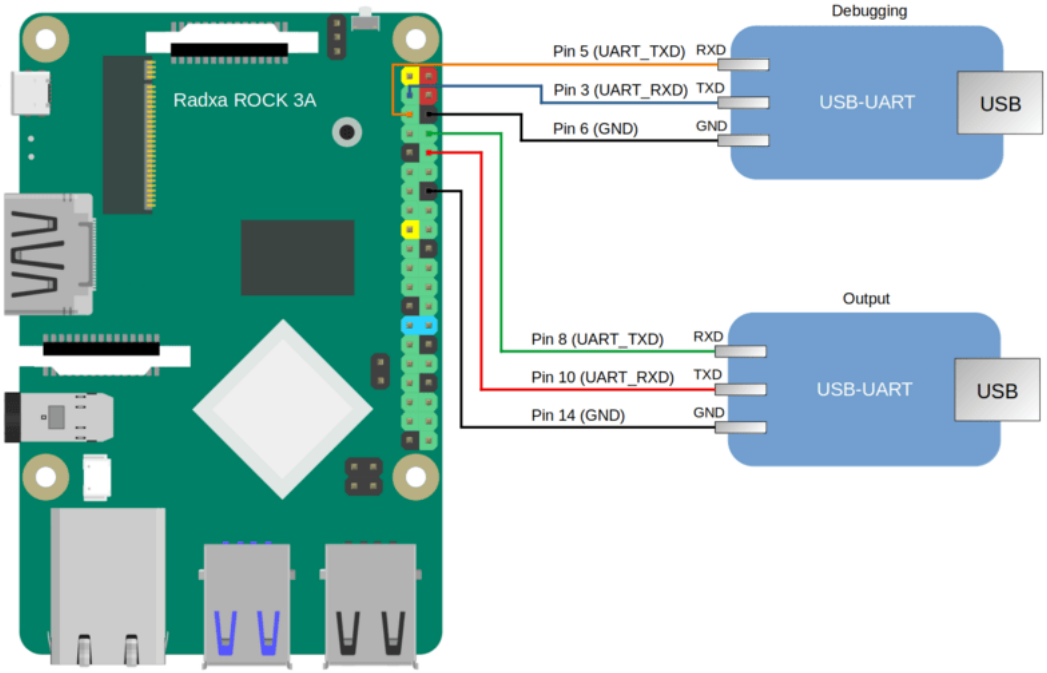


Diagram for connecting USB-UART converters and Radxa ROCK 3A

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

3. Install PuTTY or another equivalent program. Configure the settings as follows: `bps = 1500000, data bits = 8, stop bits = 1, parity = none, flow control = none`. Define the USB port connected to the USB-UART converter used for receiving output from Radxa ROCK 3A.

To allow a computer and Radxa ROCK 3A to interact through Ethernet, perform the following actions:

1. Connect the network cards of the computer and Radxa ROCK 3A to a switch or to each other.

2. Configure the computer's network card so that its IP address is in the same subnet as the IP address of the Radxa ROCK 3A network card (the settings of the Radxa ROCK 3A network card are defined in the `dhcpcd.conf` file, which is found at the path `<example name>/resources/...`).

Debugging programs for Radxa ROCK 3A

To debug programs running on the Radxa ROCK 3A, you must do the following:

1. Include a second USB-UART converter (see the figure above).

2. In the home directory of the user, create the `.gdbinit` file and add the following strings to it:

```
set sysroot /opt/KasperskyOS-Community-Edition-<version>/sysroot-aarch64-kos
add-symbol-file <path_to_debuggee>/build/einit/EinitQemu-kss/ksm.module
set follow-fork-mode parent
set follow-exec-mode same
set detach-on-fork off
set schedule-multiple on
set serial baud 115200
target extended-remote /dev/ttyUSB[n]
```

3. In the `CmakeLists.txt` file in the `<path_to_debuggee>/einit` directory, add the `GDBSTUB_KERNEL` parameter to the `build_kos_hw_image ()` command call.

4. Build the program. After startup and initialization, the entry `[KDBG ] Waiting for GDB connection infinitely` will appear in the output. The application will stop while it waits for the debugger to connect.

5. To connect the debugger, you must run gdb from the SDK: `/opt/KasperskyOS-Community-Edition-<version>/toolchain/bin/aarch64-kos-gdb`.

6. After the debugger starts, the entry `[KDBG ] Connection to GDB was established` will appear in the output.

   For more details, refer to Preparations for debugging on the hardware platform and Initial steps of debugging on the hardware platform.

Preparing a bootable SD card for Radxa ROCK 3A

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

```
# In the following command, path_to_img is the path to the image file
# [X] is the final character in the name of the SD card block device.
$ sudo pv -L 32M path_to_img | sudo dd bs=64k of=/dev/sd[X] conv=fsync
```

If `kos-image` was created when building the example, the SD card requires additional preparations before you can write the image to it. A bootable SD card for Radxa ROCK 3A can be prepared automatically or manually. After preparing the SD card, you must copy the `kos-image` file from the directory `<example name>/build/einit` to the bootable area (FAT32 partition) of the prepared SD card.

To automatically prepare the bootable SD card, connect the SD card to the computer and run the following commands:

```
# To create a bootable drive image file (*.img),
# run the script:
$ sudo /opt/KasperskyOS-Community-Edition-<version>/common/radxa_prepare_sdcard_image.sh
# In the following command, path_to_img is the path to the image file
# of the bootable drive (this path is displayed upon completion
# of the previous command), [X] is the final character
# in the name of the SD card block device.
$ sudo pv -L 32M path_to_img | sudo dd bs=64k of=/dev/sd[X] conv=fsync
```

Clearing the Radxa ROCK 3A flash memory

In some editions of the Radxa ROCK 3A, the flash memory may contain a bootloader that is incompatible with the card that you have prepared according to the instructions provided above.

If you see the message `"SPL: failed to boot from all boot devices"` when running examples on the Radxa ROCK 3A, you must clear the Radxa ROCK 3A flash memory before running the examples.

To clear the Radxa ROCK 3A flash memory:

1. Download and install the tool named `rkdeveloptool`.

   Instructions on installing this tool are provided in the documentation: https://docs.radxa.com/en/rock3/rock3a/low-level-dev/rkdeveloptool?host-os=debian#installation-for-rkdeveloptool

2. Download the bootloader for interacting with the Radxa ROCK 3A: https://dl.radxa.com/rock3/images/loader/rk356x_spl_loader_ddr1056_v1.12.109_no_check_todly.bin

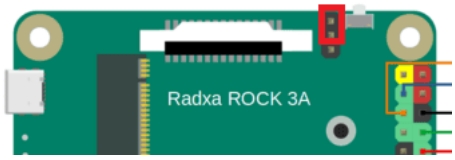
3. Switch the Radxa ROCK 3A to `Maskrom` mode:

   a. Power it off.

   b. Extract the SD card (and the eMMC module if present).

   c. Connect the computer's USB port to the ROCK 3A OTG port (upper USB 3.0 port).

   d. Connect the Radxa ROCK 3A pins as shown on the figure below and power on the Radxa ROCK 3A.



   e. Disconnect the pins that were connected at the previous step.

   f. Make sure that the Radxa ROCK 3A is in `Maskrom` mode by running the following command in the terminal:

```
$: rkdeveloptool ld
DevNo=1    Vid=0x2207,Pid=0x350a,LocationID=104    Maskrom
```

4. Copy the bootloader for RAM initialization and firmware environment preparation to the Radxa ROCK 3A by running the following command in the terminal:

```
rkdeveloptool db rk356x_spl_loader_ddr1056_v1.12.109_no_check_todly.bin
```

5. Clear the Radxa ROCK 3A flash memory by running the following commands in the terminal:

```
rkdeveloptool ef
rkdeveloptool rd
```

Running examples on Raspberry Pi 4 B or Radxa ROCK 3A

To run an example on a Raspberry Pi 4 B or Radxa ROCK 3A:

1. Go to the directory with the example and build the example.

2. Make sure that Raspberry Pi 4 B or Radxa ROCK 3A and the bootable SD card are prepared to run examples.

3. Connect the bootable SD card to the Raspberry Pi 4 B or Radxa ROCK 3A.

4. Supply power to the Raspberry Pi 4 B or Radxa ROCK 3A and wait for the example to run.

   The output displayed on the computer connected to Raspberry Pi 4 B or Radxa ROCK 3A indicates that the example started.

> Before running the examples on Radxa ROCK 3A, you also need to build the drivers provided as a source code in the SDK. Instructions for building drivers can be found in the descriptions of examples (README.md files).

> If you see the message `"SPL: failed to boot from all boot devices"` when running examples on the Radxa ROCK 3A, you must clear the Radxa ROCK 3A flash memory before running the examples. For more details, refer to Preparing Radxa ROCK 3A to run examples.

Starting processes

Overview: Einit and init.yaml

Einit initializing program

When a solution is started, the KasperskyOS kernel finds the executable file named `Einit` (initializing program) in the solution image and runs this executable file. The initializing program performs the following operations:

- Creates and starts processes when a solution is started.
- Creates IPC channels between processes when a solution is started (statically creates IPC channels).

A process of the initializing program belongs to the `Einit` class.

Generating the source code of the initializing program

The KasperskyOS SDK includes the `einit` tool, which generates the C-language source code of the initializing program. The standard way to use the `einit` tool is to integrate an einit call into one of the steps of the build script, which generates the `einit.c` file containing the source code of the initializing program. In one of the following steps of the build script, you must compile the `einit.c` file into the executable file of `Einit` and include it into the solution image.

> You are not required to create formal specification files for the initializing program. These files are provided in the KasperskyOS SDK and are automatically applied during a solution build. However, the `Einit` process class must be specified in the `security.psl` file.

The `einit` tool generates the source code of the initializing program based on the *init description*, which consists of a text file that is usually named `init.yaml`.

Syntax of init.yaml

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

- Processes that are started when the solution starts.
- IPC channels that are created when the solution starts and are used by processes to interact with each other (not with the kernel).

This data consists of a dictionary with the `entities` key containing a list of dictionaries of processes. Process dictionary keys are presented in the table below.

Process dictionary keys in an init description

Key	Required	Value
`name`	Yes	Process class name (from the EDL description).
`task`	No	Process name. If this name is not specified, the process class name will be used. Each process must have a unique name. You can start multiple processes of the same class if they have different names.
`path`	No	Name of the executable file in ROMFS (in the solution image). If this name is not specified, the process class name (without prefixes and dots) will be used. For example, processes of the `Client` and `net.Client` classes for which an executable file name is not specified will be started from the `Client` file. You can start multiple processes from the same executable file.
`connections`	No	Process IPC channel dictionaries list. This list defines the statically created IPC channels whose client IPC handles will be owned by the process. The list is empty by default. (In addition to statically created IPC channels, processes can also use dynamically created IPC channels.)
`args`	No	List of program startup parameters (`main()` function parameters). The maximum size of one item on the list is 1024 bytes.
`env`	No	Dictionary of program environment variables. The keys in this dictionary are the names of environment variables. The maximum size of an environment variable value is 65524 bytes.

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

IPC channel dictionary keys in an init description

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

Example init descriptions

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

> The build system can automatically create an init description based on the `init.yaml.in` template.

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

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

init.yaml

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

Starting processes from defined executable files

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

init.yaml

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

Starting two processes from the same executable file

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

init.yaml

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

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

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

init.yaml

```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
```

Setting the startup parameters and environment variables of programs

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

init.yaml

```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
```

Starting processes using the system program ExecutionManager

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

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

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

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

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

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

ExecutionManager component usage scenario

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

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

1. Add the ExecutionManager program to a solution. To add ExecutionManager to a solution:

- Add the following commands to the CMakeLists.txt root file:

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

> The `BlobContainer` program is required for the ExecutionManager program to work properly. This program is automatically added to a solution when adding ExecutionManager.

- The ExecutionManager component is provided in the SDK as a set of static libraries and header files, and is built for a specific solution by using the CMake command `create_execution_manager_entity()` from the CMake library `execution_manager`.

  To build the ExecutionManager program, create a directory named `execution_manager` in the root directory of the project. In the new directory, create a `CMakeLists.txt` file containing the `create_execution_manager_entity()` command.

  The CMake command `create_execution_manager_entity()` takes the following parameters:

  Mandatory `ENTITY` parameter that specifies the name of the executable file for the ExecutionManager program.

  Optional parameters:

  - `DEPENDS` – additional dependencies for building the ExecutionManager program.
  - `MAIN_CONN_NAME` – name of the IPC channel for connecting to the ExecutionManager process. It must match the value of the `mainConnection` variable when calling the ExecutionManager API in the client code.
  - `ROOT_PATH` – path to the root directory for service files of the ExecutionManager program. The default root path is `"/ROOT"`.
  - `VFS_CLIENT_LIB` – name of the client transport library used to connect the ExecutionManager program to the `VFS` program.

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

- When building a solution (CMakeLists.txt file for the Einit program), add the following executable files to the solution image:
  - Executable file of the ExecutionManager program
  - Executable file of the `BlobContainer` program

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

```
target_link_libraries (<name of the CMake target for building the client> ${execution_manager_EXECMGR_PROXY})
```

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

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

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

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

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

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

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

Overview: Env program

The system program `Env` is intended for setting startup parameters and environment variables of programs. If the `Env` program is included in a solution, the processes connected via IPC channel to the `Env` process will automatically send IPC requests to this program and receive startup parameters and environment variables when these processes are started.

> Use of the `Env` system program is an outdated way of setting startup parameters and environment variables of programs. Instead, you must set the startup parameters and environment variables of programs via the `init.yaml.in` or `init.yaml` file.
> If the value of a startup parameter or environment variable of a program is defined through the `Env` program and via the `init.yaml.in` or `init.yaml` file, the value defined through the `Env` program will be applied.

To use the `Env` program in a solution, you need to do the following:

1. Develop the code of the `Env` program using the macros and functions from the header file `sysroot-*-kos/include/env/env.h` from the KasperskyOS SDK.
2. Build the executable file of the `Env` program by linking it to the `env_server` library from the KasperskyOS SDK.
3. In the init description, indicate that the `Env` process must be started and connected to other processes (`Env` acts as a server in this case). The name of the IPC channel is assigned by the `ENV_SERVICE_NAME` macro defined in the header file `env.h`.
4. Include the `Env` executable file in the solution image.

Source code of the Env program

The source code of the `Env` program utilizes the following macros and functions from the header file `env.h`:

- `ENV_REGISTER_ARGS(name,argarr)` sets the `argarr` startup parameters for the program that will run in the context of the `name` process.
- `ENV_REGISTER_VARS(name,envarr)` sets the `envarr` environment variables for the program that will run in the context of the `name` process.
- `ENV_REGISTER_PROGRAM_ENVIRONMENT(name,argarr,envarr)` sets the `argarr` startup parameters and `envarr` environment variables for the program that will run in the context of the `name` process.
- `envServerRun()` initializes the server part of the `Env` program so that it can respond to IPC requests.

Env program usage examples

Examples of using Env to set the startup parameters and environment variables of programs

> Use of the `Env` system program is an outdated way of setting startup parameters and environment variables of programs. Instead, you must set the startup parameters and environment variables of programs via the `init.yaml.in` or `init.yaml` file.
> If the value of a startup parameter or environment variable of a program is defined through the `Env` program and via the `init.yaml.in` or `init.yaml` file, the value defined through the `Env` program will be applied.

Example of setting startup parameters of a program

Source code of the Env program is presented below. When the process named `NetVfs` starts, the program passes the following two program startup parameters to this process: `-l devfs /dev devfs 0` and `-l romfs /etc romfs ro`:

env.c

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

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

    envServerRun();
    return EXIT_SUCCESS;
}
```

Example of setting environment variables of a program

Source code of the `Env` program is presented below. When the process named `Vfs3` starts, the program passes the following two program environment variables to this process: `ROOTFS=ramdisk0,0 / ext2 0` and `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;
}
```

File systems and network

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

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

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

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

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

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

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

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

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

Using multiple VFS programs

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

Adding VFS functionality to an application

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

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

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

Contents of the VFS component

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

VFS libraries

The `vfs` CMake package contains the following libraries:

- `vfs_fs` contains implementations of the devfs, ramfs and ROMFS file systems, and adds implementations of other file systems to VFS.
- `vfs_net` contains the implementation of the devfs file system and the network stack.
- `vfs_imp` contains the `vfs_fs` and `vfs_net` libraries.
- `vfs_remote` is the client transport library that converts local calls into IPC requests to VFS and receives IPC responses.
- `vfs_server` is the VFS server transport library that receives IPC requests, converts them into local calls, and sends IPC responses.
- `vfs_local` is used to include VFS functionality in a program.

VFS executable files

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

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

The `VfsRamFs` and `VfsSdCardFs` executable files include the `vfs_server`, `vfs_fs`, `vfat` and `lwext4` libraries. The `VfsNet` executable file includes the `vfs_server`, `vfs_imp` libraries.

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

Formal specification files and header files of VFS

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

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

Libc library API supported by VFS

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

The functions implemented by the `vfs_fs` and `vfs_net` libraries are presented in the table below. The `*` character denotes the functions that are optionally included in the `vfs_fs` library (depending on the library build parameters).

Functions implemented by the vfs_fs library

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

Functions implemented by the vfs_net library

read()	bind()	getsockname()	recvfrom()
readv()	listen()	gethostbyname()	recvmsg()
write()	connect()	getnetbyaddr()	send()
writev()	accept()	getnetbyname()	sendto()
fstat()	poll()	getnetent()	sendmsg()
close()	shutdown()	setnetent()	ioctl()
fcntl()	getnameinfo()	endnetent()	sysctl()
fstatvfs()	getaddrinfo()	getprotobyname()	

pipe()	freeaddrinfo()	getprotobynumber()	
futimens()	getifaddrs()	getsockopt()	
socket()	freeifaddrs()	setsockopt()	
socketpair()	getpeername()	recv()	

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

Creating an IPC channel to VFS

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

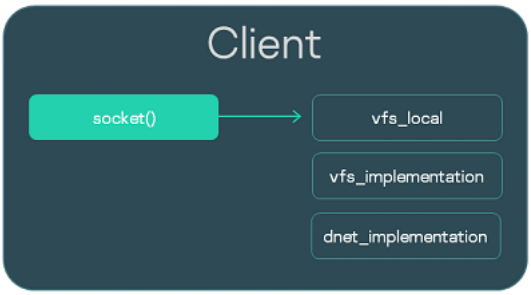
The IPC channel name must be assigned by the `_VFS_CONNECTION_ID` macro defined in the header file `sysroot-*-kos/include/vfs/defs.h` from the KasperskyOS SDK.

Init description of the example:

```
init.yaml

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

- name: VfsFsnet
```

Including VFS functionality in a program

In this example, the `Client` program includes the VFS program functionality for working with the network stack (see the figure below).



VFS component libraries in a program

The `client.c` implementation file is compiled and the `vfs_local`, `vfs_implementation` and `dnet_implementation` libraries are linked:

```
CMakeLists.txt

project (client)

include (platform/nk)

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

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

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

# Linking with VFS libraries
target_link_libraries (Client ${vfs_LOCAL_LIB} ${vfs_IMPLEMENTATION_LIB} ${dnet_IMPLEMENTATION_LIB}
```

If the `Client` program needs to use file systems, you must link the `vfs_local` and `vfs_fs` libraries, and link the libraries for implementing these file systems. In this case, you must also add a block device driver to the solution.

Overview: startup parameters and environment variables of VFS

VFS program startup parameters

- `-l <entry in fstab format>`

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

- `-f <path to fstab file>`

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

Examples of using VFS program startup parameters

Environment variables of the VFS program

- `UNMAP_ROMFS`

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

  Example of using the UNMAP_ROMFS environment variable

- `ROOTFS = <entry in fstab format>`

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

  Example of using the ROOTFS environment variable

- `VFS_CLIENT_MAX_THREADS`

The `VFS_CLIENT_MAX_THREADS` environment variable redefines the SDK configuration parameter `VFS_CLIENT_MAX_THREADS`.

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

  The `VFS_NETWORK_BACKEND` environment variable defines the VFS backend for working with the network stack. You can specify the name of the standard VFS backend: `client` (for a program that runs in the context of a client process), `server` (for a VFS program that runs in the context of a server process) or `local`, and the name of a custom VFS backend. If the `local` VFS backend is used, the name of the IPC channel is not specified (`VFS_NETWORK_BACKEND=local:`). You can specify more than one IPC channel by separating them with a comma.

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

  The `VFS_FILESYSTEM_BACKEND` environment variable defines the VFS backend for working with file systems. The name of the VFS backend and the name of the IPC channel to the VFS process are defined the same way as they are defined in the `VFS_NETWORK_BACKEND` environment variable.

Default values for startup parameters and environment variables of VFS

For the `VfsRamFs` executable file:

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

For the `VfsSdCardFs` executable file:

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

For the `VfsNet` executable file:

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

Mounting file systems when VFS starts

When the VfsRamFs and VfsSdCardFs programs start, only the RAMFS file system is mounted to the root directory by default. If you need to mount other file systems, this can be done not only by calling the `mount()` function but also by setting the startup parameters and environment variables of the VFS program.

> The `ROMFS` and `squashfs` file systems are intended for read-only operations. For this reason, you must specify the `ro` parameter to mount these file systems.

Using the startup parameter -l

One way to mount a file system is to set the startup parameter `-l <entry in fstab format>` for the VFS program.

In these examples, the devfs and ROMFS file systems will be mounted when the VFS program is started:

```
init.yaml.(in)

...
- name: VfsFirst
  args:
  - -l
  - devfs /dev devfs 0
  - -l
  - romfs /etc romfs ro
...
```

```
CMakeLists.txt

...
set_target_properties (${vfs_ENTITY} PROPERTIES
EXTRA_ARGS
"  - -l
  - devfs /dev devfs 0
  - -l
  - romfs /etc romfs ro")
...
```

Using the fstab file from the ROMFS image

When building a solution, you can add the `fstab` file to the ROMFS image. This file can be used to mount file systems by setting the startup parameter `-f <path to the fstab file>` for the VFS program.

In these examples, the file systems defined via the `fstab` file that was added to the ROMFS image during the solution build will be mounted when the VFS program is started:

```
init.yaml.(in)

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

```
CMakeLists.txt

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

Using an "external" fstab file

If the `fstab` file resides in another file system instead of in the ROMFS image, you must set the following startup parameters and environment variables for the VFS program to enable use of this file:

1. `ROOTFS`. This environment variable mounts the file system containing the `fstab` file to the root directory.

2. `UNMAP_ROMFS`. If this environment variable is defined, the `fstab` file will be sought in the file system defined through the `ROOTFS` environment variable.

3. `-f`. This startup parameter is used to mount the file systems specified in the `fstab` file.

In these examples, the ext2 file system that should contain the `fstab` file at the path `/etc/fstab` will be mounted to the root directory when the VFS program starts:

init.yaml.(in)

```
...
- name: VfsThird
  args:
  - -f
  - /etc/fstab
  env:
    ROOTFS: ramdisk0,0 / ext2 0
    UNMAP_ROMFS: 1
...
```

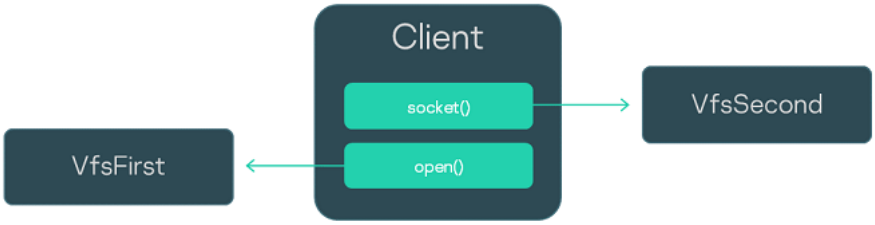
CMakeLists.txt

```
...
set_target_properties (${vfs_ENTITY} PROPERTIES
EXTRA_ARGS
"  - -f
  - /etc/fstab"
EXTRA_ENV
"  ROOTFS: ramdisk0,0 / ext2 0
  UNMAP_ROMFS: 1")
...
```

Using VFS backends to separate data streams

This example employs a secure development pattern that separates data streams related to file system use from data streams related to the use of a network stack.

The `Client` process uses file systems and the network stack. The `VfsFirst` process works with file systems, and the `VfsSecond` process provides the capability to work with the network stack. The environment variables of programs that run in the contexts of the `Client`, `VfsFirst` and `VfsSecond` processes are used to define the VFS backends that ensure the segregated use of file systems and the network stack. As a result, IPC requests of the `Client` process that are related to the use of file systems are handled by the `VfsFirst` process, and IPC requests of the `Client` process that are related to network stack use are handled by the `VfsSecond` process (see the figure below).



Process interaction scenario

Init description of the example:

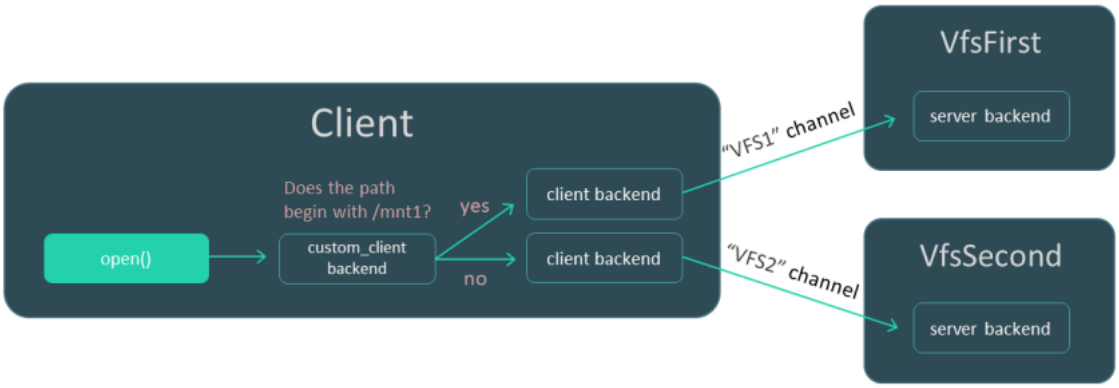
```yaml
init.yaml

entities:

- name: Client
  connections:
  - target: VfsFirst
    id: VFS1
  - target: VfsSecond
    id: VFS2
  env:
    VFS_FILESYSTEM_BACKEND: client:VFS1
    VFS_NETWORK_BACKEND: client:VFS2

- name: VfsFirst
  env:
    VFS_FILESYSTEM_BACKEND: server:VFS1

- name: VfsSecond
  env:
    VFS_NETWORK_BACKEND: server:VFS2
```

Creating a VFS backend

This example demonstrates how to create and use a custom VFS backend.

The `Client` process uses the fat32 and ext4 file systems. The `VfsFirst` process works with the fat32 file system, and the `VfsSecond` process provides the capability to work with the ext4 file system. The environment variables of programs that run in the contexts of the `Client`, `VfsFirst` and `VfsSecond` processes are used to define the VFS backends ensuring that IPC requests of the `Client` process are handled by the `VfsFirst` or `VfsSecond` process depending on the specific file system being used by the `Client` process. As a result, IPC requests of the `Client` process related to use of the fat32 file system are handled by the `VfsFirst` process, and IPC requests of the `Client` process related to use of the ext4 file system are handled by the `VfsSecond` process (see the figure below).

On the `VfsFirst` process side, the fat32 file system is mounted to the directory `/mnt1`. On the `VfsSecond` process side, the ext4 file system is mounted to the directory `/mnt2`. The custom VFS backend `custom_client` used on the `Client` process side sends IPC requests over the IPC channel `VFS1` or `VFS2` depending on whether or not the file path begins with `/mnt1`. The custom VFS backend uses the standard VFS backend `client` as an intermediary.



Process interaction scenario

Source code of the VFS backend

This implementation file contains the source code of the VFS backend `custom_client`, which uses the standard `client` VFS backends:

backend.c

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

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

#include <pthread.h>
#include <errno.h>
#include <string.h>
#include <getopt.h>
#include <assert.h>

/* Code for managing file handles */
#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;
}

/* Custom VFS backend structure */
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,
    }
};

/* Implementation of custom VFS backend methods */
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);
}

/* Custom VFS backend builder. ctx.vfs_vfat and ctx.vfs_ext4 are initialized
* as standard backends named "client". */
static struct vfs *_vfs_backend_create(Handle client_id, const char *config, int *error)
{
    (void)config;

    ctx.vfs_vfat = _vfs_init("client", client_id, "VFS1", error);
    assert(ctx.vfs_vfat != NULL && "Can't initialize client backend!");
```

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

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

    return &ctx.wrapper;
}

/* Registration of the custom VFS backend under the name 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)
```

Linking the Client program

Creating a static VFS backend library:

CMakeLists.txt

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

Linking the `Client` program to the static VFS backend library:

CMakeLists.txt

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

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

Setting the startup parameters and environment variables of programs

Init description of the example:

init.yaml

```
entities:

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

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

- name: vfs_backend.VfsSecond
```

```
  - -l
  - ahci1 /mnt2 ext4 0
  env:
    _VFS_FILESYSTEM_BACKEND: server:VFS2
```

Dynamically configuring the network stack

To change the default network stack parameters, use the `sysctl()` function or `sysctlbyname()` function that are declared in the header file `sysroot-*-kos/include/sys/sysctl.h` from the KasperskyOS SDK. The parameters that can be changed are presented in the table below.

Configurable network stack parameters

Parameter name	Parameter description
`net.inet.ip.ttl`	Maximum time to live (TTL) of sent IP packets. It does not affect the ICMP protocol.
`net.inet.ip.mtudisc`	If its value is set to `1`, "Path MTU Discovery" (RFC 1191) mode is enabled. This mode affects the maximum size of a TCP segment (Maximum Segment Size, or MSS). In this mode, the MSS value is determined by the limitations of network nodes. If "Path MTU Discovery" mode is not enabled, the MSS value does not exceed the value defined by the `net.inet.tcp.mssdflt` parameter.
`net.inet.tcp.mssdflt`	MSS value (in bytes) that is applied if only the communicating side failed to provide this value when opening the TCP connection, or if "Path MTU Discovery" mode (RFC 1191) is not enabled. This MSS value is also forwarded to the communicating side when opening a TCP connection.
`net.inet.tcp.minmss`	Minimum MSS value, in bytes.
`net.inet.tcp.mss_ifmtu`	If its value is set to `1`, the MSS value is calculated for an opened TCP connection based on the maximum size of a transmitted data block (Maximum Transmission Unit, or MTU) of the employed network interface. If its value is set to `0`, the MSS value for an opened TCP connection is calculated based on the MTU of the network interface that has the highest value for this parameter among all available network interfaces (except the loopback interface).
`net.inet.tcp.keepcnt`	Number of times to send test messages (Keep-Alive Probes, or KA) without receiving a response before the TCP connection will be considered closed. If its value is set to `0`, the number of sent keep-alive probes is unlimited.
`net.inet.tcp.keepidle`	TCP connection idle period, after which keep-alive probes begin. This is defined in conditional units, which can be converted into seconds via division by the `net.inet.tcp.slowhz` parameter value.
`net.inet.tcp.keepintvl`	Time interval between recurring keep-alive probes when no response is received. This is defined in conditional units, which can be converted into seconds via division by the `net.inet.tcp.slowhz` parameter value.
`net.inet.tcp.recvspace`	Size of the buffer (in bytes) for data received over the TCP protocol.
`net.inet.tcp.sendspace`	Size of the buffer (in bytes) for data sent over the TCP protocol.
`net.inet.udp.recvspace`	Size of the buffer (in bytes) for data received over the UDP protocol.
`net.inet.udp.sendspace`	Size of the buffer (in bytes) for data sent over the UDP protocol.

MSS configuration example:

```
static const int mss_max = 1460;
static const int mss_min = 100;
static const char* mss_max_opt_name = "net.inet.tcp.mssdflt";
static const char* mss_min_opt_name = "net.inet.tcp.minmss";
int main(void)
{
...
    if ((sysctlbyname(mss_max_opt_name, NULL, NULL, &mss_max, sizeof(mss_max)) != 0) ||
       (sysctlbyname(mss_min_opt_name, NULL, NULL, &mss_min, sizeof(mss_min)) != 0))
    {
        ERROR(START, "Can't set tcp default maximum/minimum MSS value.");
        return EXIT_FAILURE;
    }
}
```

IPC and transport

Creating IPC channels

IPC channels can be created statically or dynamically.

Statically created IPC channels

Static creation of IPC channel has the following characteristics:

- Creation of an IPC channel is performed by the parent process that starts the client and server (this is normally Einit).
- The client and server are not yet running when the IPC channel is created.
- You cannot create a new IPC channel in place of a deleted one.

The IPC channels defined in the init description are statically created. You can also use the task.h API to statically create IPC channels.

To get the client and server IPC handles and the endpoint ID (RIID), use the sl_api.h API.

Dynamically created IPC channels

Dynamic creation of IPC channels lets you change the topology of interaction between processes on the fly. For example, this is necessary if it is unknown which specific server provides the endpoint required by the client.

Dynamic creation of IPC channel has the following characteristics:

- The IPC channel is jointly created by the client and server.
- The client and server are already running when the IPC channel is created.
- You can create a new IPC channel in place of a deleted one.

To dynamically create IPC channels, use the DCM system program.

> Aside from using the DCM system program, you can also dynamically create IPC channels by using the cm_api.h and ns_api.h APIs provided by the `libkos` library together with the `NameServer` system program. However, this latter method is obsolete and therefore not recommended.

If a dynamically created IPC channel is no longer required, its client and server handles can be closed. The IPC channel can be created again if necessary.

Adding an endpoint from KasperskyOS Community Edition to a solution

To ensure that a `Client` program can use some specific functionality via the IPC mechanism, the following is required:

1. In KasperskyOS Community Edition, find the executable file (we'll call it `Server`) that implements the necessary functionality. (The term "functionality" used here refers to one or more endpoints that have their own IPC interfaces.)

2. Include the CMake package containing the `Server` file and its client library.

3. Add the `Server` executable file to the solution image.

4. Edit the init description so that when the solution starts, the `Einit` program starts a new server process from the `Server` executable file and connects it, using an IPC channel, to the process started from the `Client` file.

   > You must indicate the correct name of the IPC channel so that the transport libraries can identify this channel and find its IPC handles. The correct name of the IPC channel normally matches the name of the server process class. VFS is an exception in this case.

5. Edit the PSL description to allow startup of the server process and IPC interaction between the client and the server.

6. In the source code of the `Client` program, include the server methods header file.

7. Link the `Client` program with the client library.

Example of adding a GPIO driver to a solution

KasperskyOS Community Edition includes a `gpio_hw` file that implements GPIO driver functionality.

The following commands connect the gpio CMake package:

.\CMakeLists.txt

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

The `gpio_hw` executable file is added to a solution image by using the `gpio_HW_ENTITY` variable, whose name can be found in the configuration file of the package at `/opt/KasperskyOS-Community-Edition-<version>/sysroot-*-kos/lib/cmake/gpio/gpio-config.cmake:`

einit\CMakeLists.txt

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

The following strings need to be added to the init description:

init.yaml.in

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

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

The following strings need to be added to the PSL description:

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()
}
...
```

In the code of the `Client` program, you need to include the header file in which the GPIO driver methods are declared:

client.c

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

Finally, you need to link the `Client` program with the GPIO client library:

client\CMakeLists.txt

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

> To ensure correct operation of the GPIO driver, you may need to add the BSP component to the solution. To avoid overcomplicating this example, BSP is not examined here. For more details, see the gpio_output example: `/opt/KasperskyOS-Community-Edition-<version>/examples/gpio_output`

Creating and using your own endpoints

Overview: IPC message structure

In KasperskyOS, all interactions between processes have statically defined types. The permissible structures of an IPC message are defined by the IDL descriptions of servers.

An IPC message (request and response) contains a constant part and an (optional) arena.

Constant part of an IPC message

The *constant part of an IPC message* contains the RIID, MID, and (optionally) fixed-size parameters of interface methods.

Fixed-size parameters are parameters that have IDL types of a fixed size.

The RIID and MID identify the interface and method being called:

- The RIID (Runtime Implementation ID) is the sequence number of the utilized endpoint within the set of server endpoints (starting at zero).
- The MID (Method ID) is the sequence number of the called method within the set of methods of the utilized endpoint (starting at zero).

The type of the constant part of the IPC message is generated by the NK compiler based on the IDL description of the interface. A separate structure is generated for each interface method. `Union` types are also generated for storing any request to a process, component or interface. For more details, refer to Example generation of transport methods and types.

IPC message arena

An *IPC message arena* (hereinafter also referred to as an *arena*) contains variable-size parameters of interface methods (and/or elements of these parameters).

Variable-size parameters are parameters that have IDL types of a variable size.

For more details, refer to Working with an IPC message arena.

Maximum IPC message size

The maximum size of an IPC message is determined by the KasperskyOS kernel parameters. On most hardware platforms supported by KasperskyOS, the cumulative size of the constant part and arena of an IPC message cannot exceed 4, 8, or 16 MB.

IPC message structure verification by the security module

Prior to querying IPC message-related rules, the Kaspersky Security Module verifies that the sent IPC message is correct. Requests and responses are both validated. If the IPC message has an incorrect structure, it will be rejected without calling the security model methods associated with it.

Implementation of IPC interaction

To make it easier for a developer to implement IPC interaction, KasperskyOS Community Edition provides the following:

- NK compiler that generates transport methods and types.
- The `libkos` library that provides the API for working with IPC transport.

Implementation of simple IPC interaction is demonstrated in the echo and ping examples (`/opt/KasperskyOS-Community-Edition-<version>/examples/`).

Example generation of transport methods and types

When building a solution, the NK compiler uses the EDL, CDL and IDL descriptions to generate a set of special methods and types that simplify the creation, forwarding, receipt and processing of IPC messages.

As an example, we will examine the `Server` process class that provides the `FS` endpoint, which contains a single `Open()` method:

Server.edl

```
task class Server

/* OpsComp is the named instance of the Operations component */
components {
    OpsComp: Operations
}
```

Operations.cdl

```
component Operations

/* FS is the local name of the endpoint implementing the Filesystem interface */
endpoints {
    FS: Filesystem
}
```

Filesystem.idl

```
package Filesystem

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

These descriptions will be used to generate the files named `Server.edl.h`, `Operations.cdl.h`, and `Filesystem.idl.h`, which contain the following methods and types:

Methods and types that are common to the client and server

- **Abstract interfaces containing the pointers to the implementations of the methods included in them.**

  In our example, one abstract interface (`Filesystem`) will be generated:

```
typedef struct Filesystem {
            const struct Filesystem_ops *ops;
        } Filesystem;

typedef nk_err_t
Filesystem_Open_fn(struct Filesystem *, const
                   struct Filesystem_Open_req *,
                   const struct nk_arena *,
                   struct Filesystem_Open_res *,
                   struct nk_arena *);
```

```
typedef struct Filesystem_ops {
            Filesystem_Open_fn *Open;
        } Filesystem_ops;
```

- **Set of interface methods.**

  When calling an interface method, the corresponding values of the RIID and MID are automatically inserted into the request.

  In our example, a single `Filesystem_Open` interface method will be generated:

```
nk_err_t Filesystem_Open(struct Filesystem *self,
                         struct Filesystem_Open_req *req,
                         const
                         struct nk_arena *req_arena,
                         struct Filesystem_Open_res *res,
                         struct nk_arena *res_arena)
```

Methods and types used only on the client

- **Types of proxy objects.**

  A proxy object is used as an argument in an interface method. In our example, a single `Filesystem_proxy` proxy object type will be generated:

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

- **Functions for initializing proxy objects.**

  In our example, the single initializing function `Filesystem_proxy_init` will be generated:

```
void Filesystem_proxy_init(struct Filesystem_proxy *self,
                           struct nk_transport *transport,
                           nk_iid_t iid)
```

- **Types that define the structure of the constant part of a message for each specific method.**

  In our example, two such types will be generated: `Filesystem_Open_req` (for a request) and `Filesystem_Open_res` (for a response).

```
typedef struct __nk_packed Filesystem_Open_req {
            __nk_alignas(8)
            struct nk_message base_;
            __nk_alignas(4) nk_ptr_t name;
        } Filesystem_Open_req;

typedef struct Filesystem_Open_res {
            union {
                struct {
                    __nk_alignas(8)
                    struct nk_message base_;
                    __nk_alignas(4) nk_uint32_t h;
                };
                struct {
                    __nk_alignas(8)
                    struct nk_message base_;
                    __nk_alignas(4) nk_uint32_t h;
                } res_;
                struct Filesystem_Open_err err_;
```

```
            };
        } Filesystem_Open_res;
```

Methods and types used only on the server

- **Type containing all endpoints of a component, and the initializing function. (For each server component.)**

  > If there are embedded components, this type also contains their instances, and the initializing function takes their corresponding initialized structures. Therefore, if embedded components are present, their initialization must begin with the most deeply embedded component.

  In our example, the `Operations_component` structure and `Operations_component_init` function will be generated:

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

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

- **Type containing all endpoints provided directly by the server; all instances of components included in the server; and the initializing function.**

  In our example, the `Server_entity` structure and `Server_entity_init` function will be generated:

```
#define Server_entity Server_component

typedef struct Server_component {
            struct : Operations_component *OpsComp;
        } Server_component;

void Server_entity_init(struct Server_entity *self,
                            struct Operations_component *OpsComp)
```

- **Types that define the structure of the constant part of a message for any method of a specific interface.**

  In our example, two such types will be generated: `Filesystem_req` (for a request) and `Filesystem_res` (for a response).

```
typedef union Filesystem_req {
    struct nk_message base_;
    struct Filesystem_Open_req Open;
};

typedef union Filesystem_res {
    struct nk_message base_;
    struct Filesystem_Open_res Open;
};
```

- **Types that define the structure of the constant part of a message for any method of any endpoint of a specific component.**

  > If embedded components are present, these types also contain structures of the constant part of a message for any method of any endpoint included in all embedded components.

  In our example, two such types will be generated: `Operations_component_req` (for a request) and `Operations_component_res` (for a response).

```
typedef union Operations_component_req {
            struct nk_message base_;
            Filesystem_req FS;
        } Operations_component_req;

typedef union Operations_component_res {
            struct nk_message base_;
            Filesystem_res FS;
        } Operations_component_res;
```

- **Types that define the structure of the constant part of a message for any method of any endpoint of a specific component whose instance is included in the server.**

> If embedded components are present, these types also contain structures of the constant part of a message for any method of any endpoint included in all embedded components.

In our example, two such types will be generated: `Server_entity_req` (for a request) and `Server_entity_res` (for a response).

```
#define Server_entity_req Server_component_req

typedef union Server_component_req {
            struct nk_message base_;
            Filesystem_req OpsComp_FS;
        } Server_component_req;

#define Server_entity_res Server_component_res

typedef union Server_component_res {
            struct nk_message base_;
            Filesystem_res OpsComp_FS;
        } Server_component_res;
```

- **Dispatch methods (dispatchers) for a separate interface, component, or process class.**

  Dispatchers analyze the received query (the RIID and MID values), call the implementation of the corresponding method, and then save the response in the buffer. In our example, three dispatchers will be generated: `Filesystem_interface_dispatch`, `Operations_component_dispatch`, and `Server_entity_dispatch`.

  The process class dispatcher handles the request and calls the methods implemented by this class. If the request contains an incorrect RIID (for example, an RIID for a different endpoint that this process class does not have) or an incorrect MID, the dispatcher returns `NK_EOK` or `NK_ENOENT`.

```
nk_err_t Server_entity_dispatch(struct Server_entity *self,
                                const
                                struct nk_message *req,
                                const
                                struct nk_arena *req_arena,
                                struct nk_message *res,
                                struct nk_arena *res_arena)
```

In special cases, you can use dispatchers of the interface and the component. They take an additional argument: interface implementation ID (`nk_iid_t`). The request will be handled only if the passed argument and RIID from the request match, and if the MID is correct. Otherwise, the dispatchers return `NK_EOK` or `NK_ENOENT`.

```
nk_err_t Operations_component_dispatch(struct Operations_component *self,
                                       nk_iid_t iidOffset,
                                       const
                                       struct nk_message *req,
                                       const
                                       struct nk_arena *req_arena,
                                       struct nk_message *res,
                                       struct nk_arena *res_arena)
```

```
nk_err_t Filesystem_interface_dispatch(struct Filesystem *impl,
                                       nk_iid_t iid,
                                       const
                                       struct nk_message *req,
                                       const
                                       struct nk_arena *req_arena,
                                       struct nk_message *res,
                                       struct nk_arena *res_arena)
```

Working with an IPC message arena

Arena overview

From the perspective of a developer of KasperskyOS-based solutions, an IPC message arena is a byte buffer in the memory of a process intended for storing variable-size data transmitted over IPC. This variable-size data includes input parameters, output parameters, and error parameters of interface methods (and/or elements of these parameters) that have variable-size IDL types. An arena is also used when querying the Kaspersky Security Module to store input parameters of security interface methods (and/or elements of these parameters) that have variable-size IDL types. (Parameters of fixed-size interface methods are stored in the constant part of an IPC message.) Arenas are used on the client side and on the server side. One arena is intended either for transmitting or for receiving variable-size data through IPC, but not for both transmitting and receiving this data at the same time. In other words, arenas can be divided into IPC request arenas (containing input parameters of interface methods) and IPC response arenas (containing output parameters and error parameters of interface methods).

Only the utilized part of an arena that is occupied by data is transmitted over IPC. (If it has no data, the arena is not transmitted.) The utilized part of an arena includes one or more segments. One segment of an arena stores an array of same-type objects, such as an array of single-byte objects or an array of structures. Arrays of different types of objects may be stored in different segments of an arena. The starting address of an arena must be equal to the boundary of a *2^N*-byte sequence, where *2^N* is a value that is greater than or equal to the size of the largest primitive type in the arena (for example, the largest field of a primitive type in a structure). The address of an arena chunk must also be equal to the boundary of a *2^N*-byte sequence, where *2^N* is a value that is greater than or equal to the size of the largest primitive type in the arena chunk.

An arena must have multiple segments if the interface method has multiple variable-size input, output, or error parameters, or if multiple elements of input, output, or error parameters of the interface method have a variable size. For example, if an interface method has an input parameter of the `sequence` IDL type and an input parameter of the `bytes` IDL type, the IPC request arena will have at least two segments. In this case, it may even require additional segments if a parameter of the `sequence` IDL type consists of elements of a variable-size IDL type (for example, if elements of a sequence are `string` buffers). As another example, if an interface method has one output parameter of the `struct` IDL type that contains two fields of the `bytes` and `string` type, the IPC response arena will have two segments.

Due to the alignment of arena chunk addresses, there may be unused intervals between these chunks. Therefore, the size of the utilized part of an arena may exceed the size of the data it contains.

API for working with an arena

The set of functions and macros for working with an arena is defined in the header file `sysroot-*-kos/include/nk/arena.h` from the KasperskyOS SDK. In addition, the function for copying a string to an arena is declared in the header file `sysroot-*-kos/include/coresrv/nk/transport-kos.h` from the KasperskyOS SDK.

> You should not use functions from the header file `sysroot-*-kos/include/nk/arena.h` whose names begin with the `_` or `__` character because these functions are internal implementation details that may change.
> On hardware platforms running an Arm processor architecture, input and output parameters of the API cannot be saved in "Device memory" because this could lead to unpredictable behavior. API parameters must be saved in "Normal memory". To copy data from "Device memory" to "Normal memory" and vice versa, you must use the `RtlPedanticMemcpy()` function declared in the header file `sysroot-*-kos/include/rtl/string_pedantic.h` from the KasperskyOS SDK.

Information on the functions and macros defined in the header file `sysroot-*-kos/include/nk/arena.h` is provided in the table below. In these functions and macros, an arena and arena chunk are identified by an arena descriptor (the `nk_arena` type) and an arena chunk descriptor (the `nk_ptr_t` type), respectively. An *arena descriptor* is a structure containing three pointers: one pointer to the start of the arena, one pointer to the start of the unused part of the arena, and one pointer to the end of the arena. An *arena chunk descriptor* is a structure containing the offset of the arena chunk in bytes (relative to the start of the arena) and the size of the arena chunk in bytes. (The arena chunk descriptor type is defined in the header file `sysroot-*-kos/include/nk/types.h` from the KasperskyOS SDK.)

Creating an arena

To pass variable-size parameters of interface methods over IPC, you must create arenas on the client side and on the server side. (When IPC requests are handled on the server side using the `NkKosDoDispatch()` or `NkKosDoDispatchEx()` functions declared in the header file `sysroot-*-kos/include/coresrv/nk/transport-kos-dispatch.h` from the KasperskyOS SDK, the IPC request arena and IPC response arena are created automatically.)

To create an arena, you must create a buffer (in the stack or heap) and initialize the arena descriptor.

The address of the buffer must be aligned to comply with the maximum size of a primitive type that can be put into this buffer. The address of a dynamically created buffer usually has adequate alignment to hold the maximum amount of data of the primitive type. To ensure the required alignment of the address of a statically created buffer, you can use the `alignas` specifier.

To initialize an arena descriptor using the pointer to an already created buffer, you must use an API function or macro:

- `NK_ARENA_INITIALIZER()` macro
- `nk_arena_init()` function
- `nk_arena_create()` function
- `NK_ARENA_FINAL()` macro
- `nk_arena_init_final()` macro

The type of pointer makes no difference because this pointer is converted into a pointer to a single-byte object in the code of API functions and macros.

The `NK_ARENA_INITIALIZER()` macro and the `nk_arena_init()` and `nk_arena_create()` functions initialize such arena descriptor that may contain one or more segments. The `NK_ARENA_FINAL()` and `nk_arena_init_final()` macros initialize the descriptor of this arena, which contains only one chunk spanning the entire arena throughout its entire lifetime.

To create a buffer in the stack and initialize the arena descriptor in one step, use the `NK_ARENA_AUTO()` macro. This macro creates an arena that may contain one or more segments, and the address of the buffer created by this macro has adequate alignment to hold the maximum amount of data of the primitive type.

The size of an arena must be sufficient to hold variable-size parameters for IPC requests or IPC responses of one interface method or a set of interface methods corresponding to one interface, component, or process class while accounting for the alignment of segment addresses. Automatically generated transport code (the header files `*.idl.h`, `*.cdl.h`, and `*.edl.h`) contain `*_arena_size` constants whose values are guaranteed to comply with sufficient sizes of arenas in bytes.

The header files `*.idl.h`, `*.cdl.h`, and `*.edl.h` contain the following `*_arena_size` constants:

- `<interface name>_<interface method name>_req_arena_size` – size of an IPC request arena for the specified interface method of the specified interface
- `<interface name>_<interface method name>_res_arena_size` – size of an IPC response arena for the specified interface method of the specified interface
- `<interface name>_req_arena_size` – size of an IPC request arena for any interface method of the specified interface
- `<interface name>_res_arena_size` – size of an IPC response arena for any interface method of the specified interface

The header files `*.cdl.h` and `*.edl.h` also contain the following `*_arena_size` constants:

- `<component name>_component_req_arena_size` – size of an IPC request arena for any interface method of the specified component
- `<component name>_component_res_arena_size` – size of an IPC response arena for any interface method of the specified component

The `*.edl.h` header files also contain the following `*_arena_size` constants:

- `<process class name>_entity_req_arena_size` – size of an IPC request arena for any interface method of the specified process class
- `<process class name>_entity_res_arena_size` – size of an IPC response arena for any interface method of the specified process class

Constants containing the size of an IPC request arena or IPC response arena for one interface method (`<interface name>_<interface method name>_req_arena_size` and `<interface name>_<interface method name>_res_arena_size`) are intended for use on the client side. All other constants can be used on the client side and on the server side.

Examples of creating an arena:

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

/* Example 2 */
struct nk_arena res_arena;
char res_buf[kl_rump_DhcpcdConfig_GetOptionNtpServers_res_arena_size];
nk_arena_init(&res_arena, res_buf, res_buf + sizeof(res_buf));

/* Example 3 */
char req_buffer[kl_CliApplication_Run_req_arena_size];
struct nk_arena req_arena = nk_arena_create(req_buffer, sizeof(req_buffer));
```

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

/* Example 5 */
const char *path = "path_to_file";
size_t len = strlen(path);
/* Structure for saving the constant part of an IPC request */
struct kl_VfsFilesystem_Rmdir_req req;
struct nk_arena req_arena;
nk_arena_init_final(&req_arena, &req.path, path, len);

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

Adding data to an arena before transmission over IPC

Before transmitting an IPC request on the client side or an IPC response on the server side, data must be added to the arena. If the `NK_ARENA_FINAL()` macro or the `nk_arena_init_final()` macro is used to create an arena, you do not need to reserve an arena chunk. Instead, you only need to add data to this chunk. If the `NK_ARENA_INITIALIZER()` or `NK_ARENA_AUTO()` macro, or the `nk_arena_init()` or `nk_arena_create()` function is used to create an arena, one or multiple segments in the arena must be reserved to hold data. To reserve an arena chunk, you must use an API function or macro:

- `nk_arena_alloc_aligned()` macro
- `nk_arena_alloc()` macro
- `nk_arena_store_aligned()` macro
- `nk_arena_store()` macro
- `NkKosCopyStringToArena()` function

The arena chunk descriptor, which is passed through the output parameter of these macros and functions and through the output parameter of the `NK_ARENA_FINAL()` and `nk_arena_init_final()` macros, must be put into the constant part or into the arena of an IPC message. If an interface method has a variable-size parameter, the constant part of IPC messages contains arena chunk descriptor containing the parameter instead of the actual parameter. If an interface method has a fixed-size parameter with variable-size elements, the constant part of IPC messages contains arena chunk descriptors containing the parameter elements instead of the actual parameter elements. If an interface method has a variable-size parameter containing variable-size elements, the constant part of IPC messages contains arena chunk descriptor containing the descriptors of other arena chunks that contain these parameter elements.

The `nk_arena_store_aligned()` and `nk_arena_store()` macros and the `NkKosCopyStringToArena()` function not only reserve an arena chunk, but also copy data to this chunk.

The `nk_arena_alloc_aligned()` and `nk_arena_alloc()` macros get the address of a reserved arena chunk. You can also get an arena chunk address by using the `nk_arena_get()` macro, which additionally uses the output parameter to pass the quantity of objects accommodated in this chunk.

A reserved arena chunk can be reduced. To do so, you need to use the `nk_arena_shrink()` macro.

To undo a current reservation of arena chunks so that new chunks can be reserved for other data (after sending an IPC message), call the `nk_arena_reset()` function. If the `NK_ARENA_FINAL()` macro or `nk_arena_init_final()` macro is used to create an arena, you do not need to undo a segment reservation because the arena contains one segment spanning the entire arena throughout its entire life cycle.

Examples of adding data to an arena:

```
/* Example 1 */
char req_buffer[kl_rump_NpfctlFilter_TableAdd_req_arena_size];
struct nk_arena req_arena = NK_ARENA_INITIALIZER(req_buffer, req_buffer + sizeof(req_buffer));
/* Structure for saving the constant part of an IPC request */
struct kl_rump_NpfctlFilter_TableAdd_req req;
if (nk_arena_store(char, &req_arena, &req.tid, tid, tidlen))
    return ENOMEM;
if (nk_arena_store(char, &req_arena, &req.cidrAddr, cidr_addr, cidr_addrlen))
    return ENOMEM;

/* Example 2 */
char req_arena_buf[StringMaxSize];
struct nk_arena req_arena = NK_ARENA_INITIALIZER(req_arena_buf,
                                        req_arena_buf + sizeof(req_arena_buf));
/* Structure for saving the constant part of an IPC request */
kl_drivers_FBConsole_SetFont_req req;
size_t buf_size = strlen(fileName) + 1;
char *buf = nk_arena_alloc(char, &req_arena, &req.fileName, buf_size);
memcpy(buf, fileName, buf_size);

/* Example 3 */
char reqArenaBuf[kl_core_DCM_req_arena_size];
struct nk_arena reqArena
    = NK_ARENA_INITIALIZER(reqArenaBuf,
                           reqArenaBuf + sizeof(reqArenaBuf));
/* Structure for saving the constant part of an IPC request */
kl_core_DCM_Subscribe_req req;
rc = NkKosCopyStringToArena(&reqArena, &req.endpointType, endpointType);
if (rc != rcOk)
    return rc;
rc = NkKosCopyStringToArena(&reqArena, &req.endpointName, endpointName);
if (rc != rcOk)
    return rc;
rc = NkKosCopyStringToArena(&reqArena, &req.serverName, serverName);
if (rc != rcOk)
    return rc;

/* Example 4 */
unsigned counter = 0;
nk_ptr_t *paths;
/* Reserve an arena chunk for descriptors of other arena chunks */
paths = nk_arena_alloc(nk_ptr_t, resArena, &res->logRes, msgCount);
while(...)
{
    ...
    /* Reserve arena chunks and save their descriptors in
     * a previously reserved arena chunk with the paths address */
    char *str = nk_arena_alloc(
                                        char,
                                        resArena,
                                        &paths[counter],
                                        stringLength + 1);
    if (str == NK_NULL)
        return NK_ENOMEM;
    snprintf(str, (stringLength + 1), "%s", buffer);
    ...
    counter++;
}
```

Retrieving data from an arena after receipt over IPC

Prior to receiving an IPC request on the server side or an IPC response on the client side for an arena that will store the data received over IPC, you must undo the current reservation of segments by calling the `nk_arena_reset()` function. This must be done even if the `NK_ARENA_FINAL()` macro or the `nk_arena_init_final()` macro is used to create the arena. (The `NK_ARENA_INITIALIZER()` and `NK_ARENA_AUTO()` macros, and the `nk_arena_init()` and `nk_arena_create()` functions create an arena without reserved segments. You do not need to call the `nk_arena_reset()` function if this arena will only be used once to save data received over IPC.)

You do not need to free the currently reserved arena chunks if one of the syscalls.h API functions is directly used to receive an IPC message. However, you must do that prior to calling transport code methods on the client side and the `nk_transport_recv()` function on the server side. (The `nk_transport_recv()` function is declared in the header file `sysroot-*-kos/include/nk/transport.h` from the KasperskyOS SDK.)

When calling the `NkKosTransport_Dispatch()` function declared in the header file `sysroot-*-kos/include/coresrv/nk/transport-kos.h` from the KasperskyOS SDK, you must specify arenas that do not have reserved chunks. If the `NK_ARENA_FINAL()` or `nk_arena_init_final()` macro is used to create these arenas, call the `nk_arena_reset()` function to free the reserved chunks.

To get pointers to arena chunks and the quantity of objects accommodated in these chunks, you must use the `nk_arena_get()` macro while using the input parameter to pass the corresponding arena chunk descriptors received from the constant part and arena of the IPC message.

Example of receiving data from an arena:

```
struct nk_arena res_arena;
char res_buf[kl_rump_DhcpcdConfig_Version_res_ver_size];
nk_arena_init(&res_arena, res_buf, res_buf + sizeof(res_buf));
/* Structure for saving an IPC request */
struct kl_rump_DhcpcdConfig_Version_req req;
req.buflen = buflen;
/* Structure for saving an IPC response */
struct kl_rump_DhcpcdConfig_Version_res res;
/* Call the interface method */
if (kl_rump_DhcpcdConfig_Version(dhcpcd.proxy, &req, NULL, &res, &res_arena) != NK_EOK)
    return -1;
size_t ptrlen;
char *ptr = nk_arena_get(char, &res_arena, &res.ver, &ptrlen);
memcpy(buf, ptr, ptrlen);
```

Additional capabilities of the API

To get the arena size, call the `nk_arena_capacity()` function.

To get the size of the utilized part of the arena, call the `nk_arena_allocated_size()` function.

To verify that the arena chunk descriptor is correct, use the `nk_arena_validate()` macro.

Information about API functions and macros

Functions and macros of arena.h

Function/Macro	Information about the function/macro
`NK_ARENA_INITIALIZER()`	Purpose Initializes the arena descriptor. Parameters • [in] `_start` – pointer to the start of the arena. • [in] `_end` – pointer to the end of the arena. Macro values Code that initializes the arena descriptor.
`nk_arena_init()`	Purpose Initializes the arena descriptor.

	Parameters • [out] `self` – pointer to the arena descriptor. • [in] `start` – pointer to the start of the arena. • [in] `end` – pointer to the end of the arena. Returned values N/A
`nk_arena_create()`	Purpose Creates and initializes the arena descriptor. Parameters • [in] `start` – pointer to the start of the arena. • [in] `size` – arena size in bytes. Returned values Arena descriptor.
`NK_ARENA_AUTO()`	Purpose Creates a buffer in the stack, and creates and initializes the arena descriptor. Parameters • [in] `size` – arena size in bytes. It must be defined as a constant. Macro values Arena descriptor.
`NK_ARENA_FINAL()`	Purpose Initializes the arena descriptor containing only one segment. Parameters • [out] `ptr` – pointer to the arena chunk descriptor. • [in] `start` – pointer to the start of the arena. • [in] `count` – number of objects in the arena chunk. Macro values Arena descriptor.
`nk_arena_reset()`	Purpose Resets the reservation of arena chunks. Parameters • [in,out] `self` – pointer to the arena descriptor. Returned values N/A
`nk_arena_capacity()`	Purpose Gets the size of an arena. Parameters • [in] `self` – pointer to the arena descriptor.

	Returned values Size of the arena in bytes. Additional information If the parameter has the `NK_NULL` value, it returns `0`.
`nk_arena_validate()`	Purpose Verifies that the arena chunk descriptor is correct. Parameters • [in] `type` – type of objects in the arena chunk. • [in] `arena` – pointer to the arena descriptor. • [in] `ptr` – pointer to the arena chunk descriptor. Macro values It has a value of `0` when the arena size is not zero if all of the following conditions are fulfilled: 1. The offset specified in the arena chunk descriptor does not exceed the arena size. 2. The size specified in the arena chunk descriptor does not exceed the arena size reduced by the offset specified in the arena chunk descriptor. 3. The size specified in the arena chunk descriptor is a multiple of the size of the type of objects in this arena chunk. It has a value of `0` when the arena size is zero if all of the following conditions are fulfilled: 1. The offset specified in the arena chunk descriptor is equal to zero. 2. The size specified in the arena chunk descriptor is equal to zero. It has a value of `-1` if even one of the conditions is not fulfilled (regardless of whether the arena size is zero or non-zero), or if the `ptr` parameter has the `NK_NULL` value.
`nk_arena_allocated_size()`	Purpose Gets the size of the utilized part of the arena. Parameters • [in] `self` – pointer to the arena descriptor. Returned values Size of the utilized part of the arena, in bytes. Additional information If the parameter has the `NK_NULL` value, it returns `0`.
`nk_arena_init_final()`	Purpose Initializes the arena descriptor containing only one segment. Parameters • [out] `arena` – pointer to the arena descriptor. • [out] `ptr` – pointer to the arena chunk descriptor. • [in] `start` – pointer to the start of the arena. • [in] `count` – number of objects in the arena chunk. Macro values

	N/A
`nk_arena_alloc_aligned()`	Purpose Reserves an arena chunk with the specified alignment for the defined number of objects of the specified type. Parameters • [in] `type` – type of objects in the arena chunk. • [in,out] `arena` – pointer to the arena descriptor. • [out] `ptr` – pointer to the arena chunk descriptor. • [in] `count` – number of objects in the arena chunk. • [in] `align` – value defining the arena chunk alignment. The arena chunk address can be unaligned (*align=1*) or aligned (*align=2,4,...,2^N*) to the boundary of a *2^N*-byte sequence (for example, two-byte or four-byte). Macro values It has the address of the reserved arena chunk if successful, otherwise `NK_NULL`.
`nk_arena_alloc()`	Purpose Reserves an arena chunk for the defined number of objects of the specified type. Parameters • [in] `type` – type of objects in the arena chunk. • [in,out] `arena` – pointer to the arena descriptor. • [out] `ptr` – pointer to the arena chunk descriptor. • [in] `count` – number of objects in the arena chunk. Macro values It has the address of the reserved arena chunk if successful, otherwise `NK_NULL`.
`nk_arena_store_aligned()`	Purpose Reserves an arena chunk with the specified alignment for the defined number of objects of the specified type and copies these objects to the reserved chunk. Parameters • [in] `type` – type of objects in the arena chunk. • [in,out] `arena` – pointer to the arena descriptor. • [out] `ptr` – pointer to the arena chunk descriptor. • [in] `src` – pointer to the buffer containing the objects that need to be copied to the arena chunk. • [in] `count` – number of objects in the arena chunk. • [in] `align` – value defining the arena chunk alignment. The arena chunk address can be unaligned (*align=1*) or aligned (*align=2,4,...,2^N*) to the boundary of a *2^N*-byte sequence (for example, two-byte or four-byte). Macro values It has a value of `0` if successful, otherwise it has a value of `-1`.
`nk_arena_store()`	Purpose Reserves an arena chunk for the specified number of objects of the defined type and copies these objects to the reserved chunk. Parameters

	• [in] `type` – type of objects in the arena chunk. • [in,out] `arena` – pointer to the arena descriptor. • [out] `ptr` – pointer to the arena chunk descriptor. • [in] `src` – pointer to the buffer containing the objects that need to be copied to the arena chunk. • [in] `count` – number of objects in the arena chunk. Macro values It has a value of `0` if successful, otherwise it has a value of `-1`.
`nk_arena_get()`	Purpose Gets the address of an arena chunk and the quantity of objects of the specified type in this chunk. Parameters • [in] `type` – type of objects in the arena chunk. • [in] `arena` – pointer to the arena descriptor. • [in] `ptr` – pointer to the arena chunk descriptor. • [out] `count` – pointer to the number of objects in the arena chunk. Macro values It has the address of the arena chunk if successful, otherwise `NK_NULL`. Additional information If the size of the arena chunk is not a multiple of the size of the type of objects in this chunk, it has the `NK_NULL` value.
`nk_arena_shrink()`	Purpose Reduces the size of an arena chunk. Parameters • [in] `type` – type of objects in the arena chunk. • [in,out] `arena` – pointer to the arena descriptor. • [in,out] `ptr` – pointer to the arena chunk descriptor. • [in] `count` – quantity of objects in the reduced arena chunk. Macro values It has the address of the reduced arena chunk if successful, otherwise `NK_NULL`. Additional information If the required size of the arena chunk exceeds the current size, it has the `NK_NULL` value. If the alignment of the arena chunk that needs to be reduced does not match the type of objects for which the reduced chunk is intended, it has the `NK_NULL` value. If the arena chunk that needs to be reduced is the last chunk in the arena, the freed part of this chunk will become available for reservation of subsequent chunks.

Transport code in C++

> Before reading this section, you should review the information on the IPC mechanism in KasperskyOS and the IDL, CDL, and EDL descriptions.

Implementation of interprocess interaction requires transport code, which is responsible for generating, sending, receiving, and processing IPC messages.

However, a developer of a KasperskyOS-based solution does not have to write their own transport code. Instead, you can use the special tools and libraries included in the KasperskyOS SDK. These libraries enable a solution component developer to generate transport code based on IDL, CDL and EDL descriptions related to this component.

Transport code

The KasperskyOS SDK includes the `nkppmeta` compiler for generating transport code in C++.

The `nkppmeta` compiler lets you generate transport C++ proxy objects and stubs for use by both a client and a server.

Proxy objects are used by the client to pack the parameters of the called method into an IPC request, execute the IPC request, and unpack the IPC response.

Stubs are used by the server to unpack the parameters from the IPC request, dispatch the call to the appropriate method implementation, and pack the IPC response.

Generating transport code for development in C++

The `CMake` commands add_nk_idl(), add_nk_cdl() and add_nk_edl() are used to generate transport proxy objects and stubs using the `nkppmeta` compiler when building a solution.

C++ types in the *.idl.cpp.h file

Each interface is defined in an IDL description. This description defines the interface name, signatures of interface methods, and data types for the parameters of interface methods.

The `CMake` command add_nk_idl() is used to generate transport code when building a solution. This command creates a `CMake` target for generating header files for the defined IDL file when using the `nkppmeta` compiler.

The generated header files contain a C++ representation for the interface and data types described in the IDL file, and the methods required for use of proxy objects and stubs.

The mapping of data types declared in an IDL file to C++ types are presented in the table below.

Mapping IDL types to C++ types

IDL type	C++ type
`SInt8`	`int8_t`
`SInt16`	`int16_t`
`SInt32`	`int32_t`
`SInt64`	`int64_t`
`UInt8`	`uint8_t`

UInt16	uint16_t
UInt32	uint32_t
UInt64	uint64_t
Handle	Handle (defined in sysroot-*-kos/include/handle/handletype.h)
string	std::string
union	std::variant
struct	struct
array	std::array
sequence	std::vector
bytes	std::vector<std::byte>

Working with transport code in C++

The scenarios for developing a client and server that exchange IPC messages are presented in the sections titled Statically creating IPC channels for C++ development and Dynamically creating IPC channels for C++ development.

Statically creating IPC channels for C++ development

To implement a client program that calls a method of an endpoint provided by a server program:

1. Include the generated header file (`*.edl.cpp.h`) of the client program description.
2. Include the generated header files of the descriptions of the utilized interfaces (`*.idl.cpp.h`).
3. Include the header files from the KasperskyOS SDK:
   - `sysroot-*-kos/include/kosipc/application.h`
   - `sysroot-*-kos/include/kosipc/api.h`
   - `sysroot-*-kos/include/kosipc/connect_static_channel.h`
4. Create and initialize the application object by calling the `kosipc::MakeApplicationAutodetect()` function. (You can also use the `kosipc::MakeApplication()` and `kosipc::MakeApplicationPureClient()` functions.)
5. Get the client IPC handle of the channel and the Runtime Implementation Identifier (`RIID`) by calling the `kosipc::ConnectStaticChannel()` function.

   This function gets the name of the IPC channel (from the init.yaml file) and the qualified name of the endpoint (from the CDL and EDL descriptions of the solution component).
6. Create and initialize the proxy object for the utilized endpoint by calling the `MakeProxy()` function.

To implement a server program that provides endpoints to other programs:

1. Include the generated header file `*.edl.cpp.h` containing a description of the component structure of the program, including all provided endpoints.
2. Include the header files from the KasperskyOS SDK:

- `sysroot-*-kos/include/kosipc/event_loop.h`
- `sysroot-*-kos/include/kosipc/api.h`
- `sysroot-*-kos/include/kosipc/serve_static_channel.h`

3. Create classes containing the implementations of interfaces that this program and its components provide as endpoints.
4. Initialize the application object by calling the `kosipc::MakeApplicationAutodetect()` function.
5. Create and initialize the `kosipc::components::Root` structure, which describes the component structure of the program and describes the interfaces of all endpoints provided by the program.
6. Bind fields of the `kosipc::components::Root` structure to the objects that implement the corresponding endpoints.

   Fields of the `Root` structure replicate the hierarchy of components and endpoints that are collectively defined by the CDL and EDL files.
7. Get the server IPC handle of the channel by calling the `ServeStaticChannel()` function.

   This function gets the name of the IPC channel (from the init.yaml file) and the structure created at step 5.
8. Create the `kosipc::EventLoop` object by calling the `MakeEventLoop()` function.
9. Start the loop for dispatching incoming IPC messages by calling the `Run()` method of the `kosipc::EventLoop` object.

Dynamically creating IPC channels for C++ development

Dynamic creation of an IPC channel on the client side includes the following steps:

1. Include the generated description header file (`*.edl.cpp.h`) in the client program.
2. Include the generated header files of the descriptions of the utilized interfaces (`*.idl.cpp.h`).
3. Include the header files:
   - `/opt/KasperskyOS-Community-Edition-<version>/sysroot-*-kos/include/kosipc/application.h`
   - `/opt/KasperskyOS-Community-Edition-<version>/sysroot-*-kos/include/kosipc/make_application.h`
   - `/opt/KasperskyOS-Community-Edition-<version>/sysroot-*-kos/include/kosipc/connect_dynamic_channel.h`
4. Get the pointers to the server name and the qualified name of the endpoint by using a name server (`NameServer` system program). To do so, you must connect to the name server by calling the `NsCreate()` function and find the server that provides the required endpoint by using the `NsEnumServices()` function. For more details, refer to Dynamically creating IPC channels (cm_api.h, ns_api.h).
5. Create an application object by calling the `kosipc::MakeApplicationAutodetect()` function. (You can also use the `kosipc::MakeApplication()` and `kosipc::MakeApplicationPureClient()` functions.)

6. Create a proxy object for the required endpoint by calling the `MakeProxy()` function. Use the `kosipc::ConnectDynamicChannel()` function call as the input parameter of the `MakeProxy()` function. Pass the pointers for the server name and qualified name of the endpoint obtained at step 4 to the `kosipc::ConnectDynamicChannel()` function.

After successful initialization of the proxy object, the client can call methods of the required endpoint.

Example

```
NsHandle ns;

// Connect to a name server
Retcode rc = NsCreate(RTL_NULL, INFINITE_TIMEOUT, &ns);

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

// Get pointers to the server name and qualified name of the endpoint
rc = NsEnumServices(
     ns, interfaceName, 0,
     serverName, kl_core_Types_UCoreStringSize,
     endpointName, kl_core_Types_UCoreStringSize);

// Create and initialize the application object
kosipc::Application app = kosipc::MakeApplicationAutodetect();

// Create and initialize the proxy object
auto proxy = app.MakeProxy<IDLInterface>(
     kosipc::ConnectDynamicChannel(serverName, endpointName))

// Call the method of the required endpoint
proxy->DoSomeWork();
```

Dynamic creation of an IPC channel on the server side includes the following steps:

1. Include the generated header file (`*.edl.cpp.h`) containing a description of the component structure of the server, including all provided endpoints, in the server program.

2. Include the header files:

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

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

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

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

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

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

3. Create classes containing the implementations of interfaces that the server provides as endpoints. Create and initialize the objects of these classes.

4. Create an application object by calling the `kosipc::MakeApplicationAutodetect()` function.

5. Create and initialize the `kosipc::components::Root` class object that describes the structure of components and endpoints of the server. This structure is generated from the descriptions in the CDL and EDL files.

6. Bind the `kosipc::components::Root` class object to the class objects created at step 3.

7. Create and initialize the `kosipc::EventLoop` class object that implements a loop for dispatching incoming IPC messages by calling the `MakeEventLoop()` function. Use the `ServeDynamicChannel()` function call as an input parameter of the `MakeEventLoop()` function. Pass the `kosipc::components::Root` class object created at step 5 to the `ServeDynamicChannel()` function.

8. Start the loop for dispatching incoming IPC messages in a separate thread by calling the `Run()` method of the `kosipc::EventLoop` object.

9. Create and initialize the object that implements the handler for receiving incoming requests to dynamically create an IPC channel.

   When creating an object, you can use the `kosipc::SimpleConnectionAcceptor` class, which is the standard implementation of the `kosipc::IConnectionAcceptor` interface. (The `kosipc::IConnectionAcceptor` interface is defined in the file named `/opt/KasperskyOS-Community-Edition-<version>/sysroot-*-kos/include/kosipc/connection_acceptor.h`.) In this case, the handler will implement the following logic: if the endpoint requested by the client was published on the server, the request from the client will be accepted. Otherwise, it will be rejected.

   If you need to create your own handler, you should implement your own request handling logic in the `OnConnectionRequest()` method inherited from the `kosipc::IConnectionAcceptor` interface. This method will be called by the server when it receives a request for dynamic IPC channel creation from the client.

   You must call the `ServeDynamicChannel()` function (see step 7) strictly before creating the `kosipc::SimpleConnectionAcceptor` object.

10. Create a `kosipc::EventLoop` class object that implements a loop for receiving incoming requests to dynamically create an IPC channel by calling the `MakeEventLoop()` function. Use the `ServeConnectionRequests()` function call as an input parameter of the `MakeEventLoop()` function. Pass the object created at step 9 to the `ServeConnectionRequests()` function.

    > There can only be one loop for receiving incoming requests to dynamically create an IPC channel. The loop must work in one thread. The loop for receiving incoming requests to dynamically create an IPC channel must be created after the loop for dispatching incoming IPC channels is created (see step 7).

11. Start the loop for receiving incoming requests for a dynamic connection in the current thread by calling the `Run()` method of the `kosipc::EventLoop` object.

Example

```
// Create class objects that implement interfaces
// provided by the server as endpoints
MyIDLInterfaceImp_1 impl_1;
MyIDLInterfaceImp_2 impl_2;

// Create and initialize the application object
kosipc::Application app = kosipc::MakeApplicationAutodetect();

// Create and initialize the root object that describes
// the components and endpoints of the server
kosipc::components::Root    root;

// Bind the root object to the class objects that implement the server endpoints.
// The fields of the root object repeat the description of components and endpoints
// defined collectively by the CDL and EDL files.
root.component1.endpoint1 = &impl_1;
root.component2.endpoint2 = &impl_2;
```

```
// Create and initialize the object that implements the
// loop for dispatching incoming IPC messages
kosipc::EventLoop   loopDynamicChannel = app.MakeEventLoop(ServeDynamicChannel(root));

// Start the loop for dispatching incoming IPC messages in a separate thread
std::thread     dynChannelThread(
    [&loopDynamicChannel]() {
        loopDynamicChannel.Run();
    }
);

// Create an object that implements a standard handler for receiving incoming requests
// to dynamically create an IPC channel
kosipc::SimpleConnectionAcceptor    acceptor(root);

// Create an object that implements a loop for receiving incoming requests
// to dynamically create an IPC channel
kosipc::EventLoop   loopDynamicChannel = app.MakeEventLoop(ServeConnectionRequests(&acceptor));

// Start the loop for receiving incoming requests to dynamically create an IPC channel in the current thread
loopConnectionReq.Run();
```

If necessary, you can create and initialize multiple `kosipc::components::Root` class objects combined into a list of objects of the `ServiceList` type using the `AddServices()` method. For example, use of multiple objects enables you to separate components and endpoints of the server into groups or publish endpoints under different names.

Example

```
// Create and initialize the group_1 object
kosipc::components::Root    group_1;

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

// Create and initialize the group_2 object
kosipc::components::Root    group_2;

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

// Create and initialize the group_3 object
kosipc::components::Root group_3;

group_3.component1.endpoint1 = &impl_5;

// Create a list of objects
ServiceList endpoints;
endpoints.AddServices(group_1);
endpoints.AddServices(group_2);
endpoints.AddServices(group_3.component1.endpoint1, "SomeCustomEndpointName");

// Create an object that implements the handler for receiving incoming requests
// to dynamically create an IPC channel
kosipc::SimpleConnectionAcceptor    acceptor(std::move(endpoints));

// Create an object that implements a loop for receiving incoming requests
// to dynamically create an IPC channel
kosipc::EventLoop   loopDynamicChannel = app.MakeEventLoop(ServeConnectionRequests(&acceptor));
```

Working with KPA packages

A *KPA package* is a file in the proprietary KPA format that serves as packaging for a program intended for installation in a KasperskyOS-based solution. It includes the following elements:

- KPA package header. Unique sequence of bytes used to identify the KPA format.

- KPA package manifest. Data structure that describes a JSON file containing detailed information about the KPA package.

- KPA package components. Aligned byte sequences with arbitrary contents. KPA package components may include executable files, libraries, text data, or any other data required for the program to work.

- KPA package index. Data structure that describes the number of KPA package components, their checksums, and their sizes.

A compressed KPA package is a file in KPAC format. KPAC format is a variant of the ZIP format but with the following limitations: no directory hierarchy, no embedded ZIP archives, and a limit on the size and quantity of its archived files. A KPAC file contains a KPA package, its external signature, and an index file. The external signature of a KPA package is a file in the proprietary KCAT format that resides outside of the KPA package file. The external signature protects against spoofing and modifications of the KPA package itself and the index file of the KPA package. The index file of a KPA package is a file in the proprietary KIDX format that is used to check the integrity of the KPA package.

Managing KPA packages

KasperskyOS Community Edition includes the following resources for KPA package management:

- `CMake` library platform/kpa intended for building KPA packages during the KasperskyOS-based solution build process. When using functions of the `CMake` library platform/kpa, the KPA package manifest is created automatically.

- Tools that let you build a KPA package from program source files in the system where the SDK is installed and then install the KPA package in a built KasperskyOS-based solution image.

- PackageManager component that lets you install KPA packages in an operational KasperskyOS-based solution, remove KPA packages, and receive information about them.

KPA package manifest

A *KPA package manifest* is a JSON file that contains the information required when installing and using a KPA package. A list of the main keys for the KPA package manifest is presented in the table below.

Main keys of a KPA package manifest

Key name	Value type	Description	Required	Example
`version`	String	Manifest version number in the format `{major}.{minor}`, where `major` is the major version of the manifest and `minor` is the minor version of the manifest. The current version of the manifest is 2.0. It is not recommended to change this version because any change could result in problems with program operation.	Yes	`"version": "2.0"`
`digestType`	String	Identifier (OID) of the hash function that is used to calculate the checksum of object files and other checksums. If the value is not defined or is an empty string, the value `"2.16.840.1.101.3.4.2.1"` (SHA256 hash function) is used by default.	No	`"digestType": "2.16.840.1.101.3.4.2.1"`
`application`	Object	Information about the program.	Yes	Application object
`platform`	Object	Information about the target platform of the program.	Yes	Platform object
`components`	List of objects	List of KPA package components.	No	List of "components" objects

`runConfiguration`	List of objects	List of program startup configurations.	No	List of runConfiguration objects
`privateStorage`	Object	Isolated storage of program data.	No	PrivateStorage object
`extensions`	Object	User-defined object that lets you add custom contents to the manifest to meet the specific demands of the program developer.	No	Custom contents

Application object

The `application` object includes keys containing information about the program installed from the KPA package. A list of these keys is provided in the table below.

List of keys of an application object

Key name	Value type	Description	Required	Example
`id`	String	Unique ID of the program. This ID enables unequivocal identification of installed program. After it is published, the program ID must not change. If the program ID changes, all other participants of the installed program life cycle will see it as a different program.	Yes	`"id": "helloworld"`
`name`	String	Displayed named of the program (for example, used for display in user interfaces).	Yes	`"name": "Hello World"`
`version`	String	Program version.	Yes	`"version": "1.2.34"`
`buildNumber`	String	Program build number.	Yes	`"buildNumber": "182"`
`systemApplication`	Logical	Logical value indicating whether the program is a system program: `true` means that it is a system program, and `false` means that it is application. In the current version of KasperskyOS Community Edition, the only available value is `false`.	Yes	`"systemApplication": false`
`description`	String	Program description.	No	`"description": "Sample application"`

Platform object

The `platform` object includes keys containing information about the target platform of the program. A list of these keys is provided in the table below.

List of keys of a platform object

Key name		Value type	Description	Required	Example
`id`		String	Operating system ID.	Yes	`"id": "kos.ce"`

`sdk`		Object	Target API level that must be provided by the KasperskyOS Community Edition device for optimal operation of the program, and the name and version of the SDK used to build this program.	Yes	`"sdk":`
					`{`
					`"buildSdkName": "KasperskyOS-Community-Edition-RaspberryPi4b"`
					`"buildSdkVersion": "1.3.0.3"`
					`"targetApiLevel": 4`
					`}`

	`buildSdkName`	String	Name of the SDK used to build this program.	Yes	
	`buildSdkVersion`	String	Version of the SDK used to build this program.	Yes	
	`targetApiLevel`	Integer	Target API level that must be provided by the device hosting KasperskyOS Community Edition.	Yes	

`hwSpecification`		Object	Technical specification of the hardware platform.	Yes	`"hwSpecification": { "arch": "aarch64-kos" "cpu": "bcm2711" }`
	`arch`	String	Architecture of the hardware platform.	Yes	
	`cpu`	String	Name of the processor used to run the hardware platform.	Yes	

List of "components" objects

The list of `components` objects includes keys containing information about the components added to the KPA package. A list of these keys is provided in the table below.

List of keys for describing a component instance in the list of components objects

Key name	Value type	Description	Required	Example
`name`	String	Name of the KPA package component.	Yes	`"name": "imageHighResolution"`
`directoryPath`	String	Path of the directory relative to the path `/<package name>/res` where KPA package component should be installed. It is ignored if the `componentType` key value is set to `bin` or `lib`.	No	`"directoryPath": "images/highResolution"`
`digest`	String	Checksum of the KPA package component file.	Yes	`"digest": "2d541fe063c195a3b8a90204f2c234e1b5daf66`
`componentType`	String	Component type: • `bin` – an executable file. It is put into the `/<program_name>/bin` directory when the program is installed. • `lib` – shared library. It is put into the directory `/<program_name>/lib` when the program is installed. • `res` – user-defined resource. If the `directoryPath` key value is defined, the resource is put into the directory `/<package_name>/res/<directoryPath>` when the program is installed. Otherwise, the resource is put into the directory `/<package_name>/res`. • `manifestLocale` – file containing information for KPA package manifest localization. When the program is installed, KPA package manifest localization files are put into the directory `/<program_name>/manifest_locales` if a value is not defined for the `directoryPath` key.	Yes	`"componentType": "res"`

List of runConfiguration objects

The list of `runConfiguration` objects includes keys containing information about the possible startup configurations of the program. A list of these keys is provided in the table below.

List of keys for describing a startup configuration instance for the list of runConfiguration objects

Key name	Value type	Description	Required	Example

id		String	Startup configuration ID that is unique within the specific KPA package.	Yes	"id": "app"
name		String	Startup configuration name. This is a localized string.	Yes	"name": "application" or "name": "@runConfigurationLocalizedName@"
type		String	Startup configuration type: • gui – process with a graphical user interface. • service – endpoint process.	Yes	"type": "service"
args		List of strings	List of arguments in the form of a string array.	No	"args": ["networkSpeed=4096", "nthreads=2"]

envVariables		List of objects	List of environment variables.	No	"envVariables":
					{
					"name": "IMAGES"
					"value": "images"
					}

	name	String	Environment variable name.	Yes (if envVariables is used)	
	value	String	Environment variable value (can be an empty string).	Yes (if envVariables is used)	
primary		Logical	Indicates whether the startup configuration is the primary one during program startup: true means that it is primary, and false means that it is not.	Yes	"primary": true
autorun		Logical	Indicates whether this configuration is started automatically: true means that it is started automatically, and false means that it is not. The default value is false.	No	"autorun": false
eiid		String	Program security class. Required for the KasperskyOS security module.	No	"eiid": "application.Application"
path		String	Path to the KPA package component file. The path is defined relative to /<application_id>.	Yes	"path": "bin/application"

usesService		List of objects	List of startup configurations on which this startup configuration depends. Not supported in the current version of KasperskyOS Community Edition.	No	"usesService":
					{
					"name": "database"
					"useType": "child"
					}

	name	String	Name of the utilized startup configuration. If the startup configuration is in another KPA package, the value of this field must have the format "<package_name>:<startup_configuration_name>". If the startup configuration is in this KPA package, the field may have a value in the format "<package_name>:<startup_configuration_name>" (<package_name> corresponds to the current package) or only "<startup_configuration_name>".	Yes (if usesService is used)	
	useType	String	Startup configuration usage type: • child – the program can start the specified startup configuration as a child process.	Yes (if usesService is used)	

			• `serviceCanUse` – the program may require the specified endpoint. • `serviceActive` – the specified endpoint must be active when the program is started.		

PrivateStorage object

The `privateStorage` object includes keys containing information about program data storage. A list of these keys is provided in the table below.

> The `privateStorage` object is not supported in the current version of KasperskyOS Community Edition.

List of keys of a privateStorage object

Key name	Value type	Description	Required	Example
`size`	String	Program data storage size in MB.	Yes	`"size": "512"`
`fsType`	String	Program data storage file system type.	Yes	`"fsType": "ext4"`

Tools for managing KPA packages

KasperskyOS Community Edition includes the following tools for managing KPA packages:

- cas-pack for building a KPA package in a system where the SDK is installed.
- cas-inspect for getting information about KPA package contents when working with the SDK.
- cas-pm for installing one or more KPA packages in a built KasperskyOS-based solution image.

The executable files of these tools are located in the directory `/opt/KasperskyOS-Community-Edition-<version>/toolchain/bin/`.

cas-pack tool

KasperskyOS Community Edition includes the `cas-pack` tool (the `toolchain/bin/cas-pack` executable file) intended for building a KPA package in a system where the KasperskyOS Community Edition SDK is installed.

Syntax of the shell command for running the `cas-pack` tool:

```
cas-pack {-o|--output} <FILE> --manifest <FILE> --verify [--version] [-h|--help] <FILES>...
```

Parameters:

- `{-o|--output}` `<FILE>`

  Full name of the built KPA package file.

- `--manifest <FILE>`

  Full name of the KPA package manifest file.

- `<FILES>`

  List of the full names of files that will be included in the KPA package. Use a space to separate list items. You can use the `*` character to select all files in the directory.

- `--verify`

  Verification of the presence of all KPA package components specified in its manifest and the absence of unspecified components, and calculation of the checksums of KPA package components and their comparison with those specified in the KPA package manifest.

- `--version`

  Tool version.

- `-h|--help`

  Help text.

Example of the shell command for running the `cas-pack` tool:

```
# Packages all program source files residing in the
# files directory into the KPA package named helloworld.kpa. A KPA package is built
# using information from the manifest.json file, which is the
# KPA package manifest. The built KPA package is verified.

/opt/KasperskyOS-Community-Edition-<version>/toolchain/bin/cas-pack
         --output   ./helloworld.kpa
         --manifest ./manifest.json
         --verify
         ./files/*
```

cas-inspect tool

The KasperskyOS Community Edition is delivered with the `cas-inspect` tool (`toolchain/bin/cas-inspect` executable file), which lets you get information about the contents of KPA package when working with SDK.

Syntax of the shell command for running the `cas-inspect` tool:

```
cas-inspect [-h|--help] [--version] {-i|--input} <PACKAGE> --verify
[<COMMAND>] [-o <path>]
```

Parameters:

- `{-i|--input} <PACKAGE>`

  Path to the KPA package (`*.kpa` file).

- `<COMMAND>`

  Commands:

  - `dump` – prints to standard output the KPA package manifest and information about KPA package components, which includes the size in bytes (`Size`), offset in bytes (`Offset` – relative to the end of the

KPA package manifest, and `Absolute` – relative to the start of the KPA package), and the checksum (`Digest`). This parameter is applied by default.

- `read <manifest|blobs|<hash>` – prints the KPA package manifest (`read manifest`), the contents of all KPA package components (`read blobs`), or the contents of one KPA package component with the defined checksum (`read <hash>`). When using the `-o <path>` parameter, the output is printed to a file. Otherwise, it is printed to the standard output.

- `list` – prints the checksum, offset (in bytes) relative to the start of the KPA package, and the size (in bytes) for all KPA package components to the standard output.

- `read-files <FILES>...` – prints the contents of a KPA package component based on the component file name. You can specify multiple names of KPA package component files if you separate these names with a space. When using the `-o <path>` parameter, the output is printed to a file.

- `list-files` – prints to standard output a list of all names of KPA package component files included in the KPA package manifest.

- `-o <path>`

  Path to the file or directory for saving data when using the commands `read {manifest|blobs|<hash>}` and `read-files <FILES>...`. When printing the KPA package manifest (`read manifest`) or the contents of the KPA package component with the defined checksum (`read <hash>`), you must specify the file path. When printing the contents of all program components (`read blobs`), you must specify the path to the directory where each program component will be saved in a separate file with a name matching the checksum of the specific component. When printing the contents of all KPA package components (`read-files <FILES>...`), you must specify the path to the directory where each KPA package component will be saved in a separate file with the name of the specific component.

- `--verify`

  Verification of the presence of all KPA package components specified in its manifest and the absence of unspecified KPA package components, and calculation of the checksums of KPA package components and their comparison with those specified in the KPA package manifest.

- `-h|--help`

  Help text.

- `--version`

  Tool version.

Examples of shell commands for running the `cas-inspect` tool:

```
# Prints the KPA package manifest and information about
# KPA package components.
cas-inspect -i helloworld.kpa

# Prints the KPA package manifest and information about
# KPA package components to the console, and verifies the availability of the KPA package components
# specified in the manifest and the checksums of
# KPA package components.
cas-inspect -i helloworld.kpa --verify

# Prints the KPA package manifest to a file.
cas-inspect -i helloworld.kpa read manifest -o ./manifest

# Prints the contents of the KPA package component with the defined
# checksum to a file.
cas-inspect -i helloworld.kpa read 5d8071308518a7bb003aa084fc995
d2f09b79e9e52f8cd296cb3ee2644ad3951 -o ./comp

# Prints the contents of each KPA package component to a separate file.
cas-inspect -i helloworld.kpa read blobs -o .
```

```
# Prints information about KPA package components to the console.
cas-inspect -i helloworld.kpa list
```

cas-pm tool

KasperskyOS Community Edition includes the `cas-pm` tool (the `toolchain/bin/cas-pm` executable file), which installs KPA packages in a built KasperskyOS-based solution image.

Syntax of the shell command for running the `cas-pm` tool:

```
cas-pm {-p|--pkgsdir} <DIR> {-d|--dbpath} <PATH> {-a|--appsdir} <DIR>
[--rootdir <DIR>] [{-l|--layout} <PATH>] {-e|--extention} <ARG> {-r|--reinstall}
<-v[v...]> [--sign-ext <ARG>] [--index-ext <ARG>] <PACKAGES>... [--version] [-h|--help]
```

Parameters:

- `{-p|--pkgsdir}` `<DIR>`

  SDK host system path to the directory containing the KPA packages to be installed.

- `{-d|--dbpath}` `<PATH>`

  Full name of the SQLite database file that contains data on the installed KPA packages. If the database has not yet been created, it will be automatically generated upon startup of the tool with the specified name, and information about the installed KPA packages will be added to it.

  To ensure that the PackageManager component can detect the database after startup of the KasperskyOS-based solution, complete the following steps:

  1. After calling the `cas-pm` tool, copy the database file into the file system that will be put into the KasperskyOS-based solution image. If the full name of the database file was originally specified in this file system, this step can be omitted.

  2. The full name of the database file in the file system that will be put into the KasperskyOS-based solution image must be passed to the CMake command `create_package_manager_entity()` via the `DB_PATH` parameter (for more details, see PackageManager component usage scenario).

- `{-a|--appsdir}` `<DIR>`

  SDK host system path to the directory intended for storing KPA packages before they are written to the KasperskyOS-based solution image.

- `--rootdir` `<DIR>`

  Relative directory that will be used for installing KPA packages to the KasperskyOS-based solution image. Specify the directory in the file system that will be put into the KasperskyOS-based solution image. Information about the location of KPA packages will be entered into the database and will be required by the PackageManager component when removing KPA packages.

- `{-l|--layout}` `<PATH>`

Full name of the JSON file that is used to redefine the paths for installing KPA package components. Specify the full file name in the system where the SDK is installed. By default, when a KPA package is installed, its components are put into directories depending on the specific type of KPA package component (for more details, see the `componentType` key in the article titled List of "components" objects). To change the names of the default directories, define your own values for the keys: `bin`, `res`, `lib` and `manifestLocale`. To ensure that the PackageManager component can detect KPA package components after the KasperskyOS-based solution is started, the name of this file must be passed in the `CUSTOM_LAYOUT` parameter of the CMake command `create_package_manager_entity()` (for more details, see PackageManager component usage scenario).

Example of a `custom_layout_schema.json` file:

```
{
    "bin"            : "custom-bin-path",
    "res"            : "CustomResPath",
    "lib"            : "CustomLibPath",
    "manifestLocale" : "Custom_manifestLocale_Path"
}
```

- `{-e|--extention}` `<ARG>`

  Extension for a KPA package file. The default value is `kpa`.

- `{-r|--reinstall}`

  Reinstallation of KPA packages.

- `-v[v...]`

  Log level for actions performed by the tool. The number of `v` characters indicates the log level. Messages are printed to standard output. Available values:

  - `-v`

    This level logs non-detailed information about normal operation of the tool, and errors and warnings about potential problems.

  - `-vv[v...]`

    This level adds detailed logging of tool operation information that may be useful to developers for troubleshooting.

- `--sign-ext` `<ARG>`

  Extension for a KPA package external signature file. For more details about a KPA package external signature, see Working with KPA packages.

- `--index-ext` `<ARG>`

  Extension for a KPA package index file. For more details about a KPA package index file, see Working with KPA packages.

- `<PACKAGES>`

  List of full names of installed KPA packages in the system where the SDK is installed. You do not need to specify the file extension. Use a space to separate list items.

- `--version`

  Tool version.

- `-h|--help`

  Help text.

Examples of shell commands for running the `cas-pm` tool:

```
# Reinstall the helloworld.kpa package located in the directory defined by the ${PKG_DIR} variable.
# The package will first be put into the directory ${ROOTFS_DIR}/package before it is written to the image of the
# KasperskyOS-based solution. The directory ${ROOTFS_DIR}/package resides in the file system
# that will be copied to the solution image. The repository.sqlite database containing information about the
# packages installed in the image will reside in the directory defined by the ${ROOTFS_DIR} variable.
# During installation, the actions performed by the tool are logged in detail.
/opt/KasperskyOS-Community-Edition-<version>/toolchain/bin/cas-pm
                            --pkgsdir   ${PKG_DIR}
                            --reinstall
                            -vvv
                            --dbpath    ${ROOTFS_DIR}/repository.sqlite
                            --appsdir   ${ROOTFS_DIR}/package
                            --rootdir   ${ROOTFS_DIR}/package
                            --extension kpa
                             ${PKG_DIR}/helloworld
```

PackageManager component usage scenario

The PackageManager component provides an API for managing KPA packages in solutions that are based on KasperskyOS.

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

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

Adding the PackageManager component to a KasperskyOS-based solution

Hereinafter the "client" refers to the program that uses the PackageManager component API to manage KPA packages.

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

1. Add the PackageManager program to a solution. To add PackageManager to a solution:

   - Add the following commands to the CMakeLists.txt root file:

   ```
   find_package (package_manager REQUIRED)
   include_directories (${package_manager_INCLUDE})
   add_subdirectory (package_manager)
   ```

   - The PackageManager component is provided in the SDK as a set of static libraries and header files, and is built for a specific solution by using the CMake command `create_package_manager_entity()` from the CMake library `package_manager`.

     To build the PackageManager program, create a directory named `package_manager` in the root directory of the project. In the new directory, create a `CMakeLists.txt` file containing the `create_package_manager_entity()` command.

     The CMake command `create_package_manager_entity()` takes the following parameters:

     Mandatory `ENTITY` parameter that specifies the name of the executable file for the PackageManager program.

Optional parameters:

- `DEPENDS` – additional dependencies for building the PackageManager program.
- `MAIN_CONN_NAME` – name of the IPC channel for connecting to the PackageManager process. It must match the value of the `mainConnection` variable when calling the PackageManager API in the client code.
- `ROOT_PATH` – path to the root directory for service files of the PackageManager program. The default value is `"/ROOT"`.
- `PKGS_DIR` – path to the directory containing the KPA packages to be installed.
- `PKG_EXTENSION` – extension for the KPA package file.
- `DB_PATH` – full name of the SQLite database file in the KasperskyOS-based solution image containing data on the installed KPA packages.
- `APPS_DIR` – path to the directory where the KPA packages will be installed.
- `VFS_CLIENT_LIB` – name of the client transport library used to connect the PackageManager program to the `VFS` program.
- `NK_MODULE_NAME` – path for installing the header files of the PackageManager component in the SDK relative to the directory `/opt/KasperskyOS-Community-Edition-<version>/sysroot-*-kos/include/`. Default value: `kl/package_manager`.
- `AUDIT_CONN_NAME` – name of the IPC channel for connecting to the `AuditStorage` process.
- `WITHOUT_SIGN_MODE` – external signature verification mode: `true` – lack of an external signature is not considered an error, `false` – lack of an external signature is considered an error. The default value is `false`.
- `MANIFEST_SCHEMA_BUILD_STORE` – path to the build directory of the KasperskyOS-based solution image containing the manifest schema.
- `MANIFEST_SCHEMA_RUNTIME_PATH` – path to the directory of the started KasperskyOS-based solution containing the manifest schema.
- `PATH_TO_ADDITIONAL_EXTENSIONS_SCHEMAS` – path to the directory containing additional manifest schemas for objects of an arbitrary format that are defined in the `extentions` key value of the KPA package manifest.
- `CUSTOM_LAYOUT` – full name of the JSON file that is used to redefine the paths for installing KPA package components.

```
include (package_manager/create_package_manager_entity)
create_package_manager_entity(
    ENTITY PkgMgrEntity
    NK_MODULE_NAME "package_manager"
    MAIN_CONN_NAME "PkgMgrEntity"
    ROOT_PATH "/"
    PKGS_DIR "/packages"
    PKG_EXTENSION "kpa"
    DB_PATH "${DB_PATH}"
    APPS_DIR "${APPS_PATH}"
    MANIFEST_SCHEMA_BUILD_STORE "${CMAKE_BINARY_DIR}/rootdir/schema"
    MANIFEST_SCHEMA_RUNTIME_PATH "/schema"
    PATH_TO_ADDITIONAL_EXTENSIONS_SCHEMAS "${CMAKE_SOURCE_DIR}/resources/additional_extensions/"
    CUSTOM_LAYOUT "/custom_layout_schema.json"
```

```
    VFS_CLIENT_LIB vfs::client
    AUDIT_CONN_NAME "audit_storage"
    WITHOUT_SIGN_MODE TRUE)
```

- When building a solution (CMakeLists.txt file for the Einit program), add the PackageManager program executable file to the KasperskyOS-based solution image:

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

```
target_link_libraries (<name of the CMake target for building the client> ${package_manager_CLIENT_LIBS})
```

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

   a. To enable the PackageManager program to manage KPA packages, the solution security policy must allow the following interactions for the `package_manager.PkgMgrEntity` process class:

      - Access to all endpoints of the `VFS` program.

      - Access to the core endpoints `Sync`, `VMM`, `Thread`, `HAL`, `Handle`, `FS`, `Notice`, `CM` and `Profiler` (their descriptions are located in the directory `sysroot-*-kos/include/kl/core` from the SDK).

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

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

4. Use of the PackageManager program API in the client code.

   Use the header file `component/package_manager/kos_ipc/package_manager_proxy.h` for this. For more details, refer to PackageManager component.

Return codes

Overview

In a KasperskyOS-based solution, the return codes of functions of various APIs (for example, APIs of the `libkos` and `kdf` libraries, drivers, transport code, and application software) are 32-bit signed integers. This type is defined in the `sysroot-*-kos/include/rtl/retcode.h` header file from the KasperskyOS SDK as follows:

```
typedef __INT32_TYPE__ Retcode;
```

The set of return codes consists of a success code with a value of `0` and error codes. An error code is interpreted as a data structure whose format is described in the `sysroot-*-kos/include/rtl/retcode.h` header file from the KasperskyOS SDK. This format provides for multiple fields that contain not only information about the results of a function call, but also the following additional information:

- Flag in the `Customer` field indicating that the error code was defined by the developers of the KasperskyOS-based solution and not by the developers of software from the KasperskyOS SDK.

  Thanks to the flag in the `Customer` field, developers of a KasperskyOS-based solution and developers of software from the KasperskyOS SDK can define error codes from non-overlapping sets.

- Global ID of the error code in the `Space` field.

  Global IDs let you define non-overlapping sets of error codes. Error codes can be generic or specific. Generic error codes can be used in the APIs of any solution components and in the APIs of any constituent parts of solution components (for example, a driver or VFS may be a constituent part of a solution component). Specific error codes are used in the APIs of one or more solution components or in the APIs of one or more constituent parts of solution components.

  For example, the `RC_SPACE_GENERAL` ID corresponds to generic errors, the `RC_SPACE_KERNEL` ID corresponds to error codes of the kernel, and the `RC_SPACE_DRIVERS` ID corresponds to error codes of drivers.

- Local ID of the error code in the `Facility` field.

  Local IDs let you define non-overlapping subsets of error codes within the set of error codes corresponding to one global ID. For example, the set of error codes with the global ID `RC_SPACE_DRIVERS` includes non-overlapping subsets of error codes with the local IDs `RC_FACILITY_I2C`, `RC_FACILITY_USB`, and `RC_FACILITY_BLKDEV`.

The global and local IDs of specific error codes are assigned by the developers of a KasperskyOS-based solution and by the developers of software from the KasperskyOS SDK independently of each other. In other words, two sets of global IDs are generated. Each global ID has a unique meaning within one set. Each local ID has a unique meaning within a set of local IDs related to one global ID. Generic error codes can be used in any API.

This type of centralized approach helps avoid situations in which the same error codes have various meanings within a KasperskyOS-based solution. This is necessary to eliminate a potential problem transmitting error codes through different APIs. For example, this problem occurs when drivers call `kdf` library functions, receive error codes, and return these codes through their own APIs. If error codes are generated without a centralized approach, the same error code can have different meanings for the `kdf` library and for the driver. Under these conditions, drivers return correct error codes only if the error codes of the `kdf` library are converted into error codes of each driver. In other words, error codes in a KasperskyOS-based solution are assigned in such way that does not require conversion of these codes during their transit through various APIs.

> The information about return codes provided here does not apply to functions of a POSIX interface or the APIs of third-party software used in KasperskyOS-based solutions.

Generic return codes

Return codes that are generic for APIs of all solution components and their constituent parts are defined in the `sysroot-*-kos/include/rtl/retcode.h` header file from the KasperskyOS SDK. Descriptions of generic return codes are provided in the table below.

Generic return codes

Return code	Description
`rcOk` (corresponds to the `0` value)	The function completed successfully.
`rcInvalidArgument`	Invalid function parameter.
`rcNotConnected`	No connection between the client and server sides of interaction. For example, there is no server IPC handle.
`rcOutOfMemory`	Insufficient memory to perform the operation.
`rcBufferTooSmall`	Insufficient buffer size.
`rcInternalError`	The function ended with an internal error related to incorrect logic. Some examples of internal errors include values outside of the permissible limits, and null indicators and values where they are not permitted.
`rcTransferError`	Error sending an IPC message.
`rcReceiveError`	Error receiving an IPC message.
`rcSourceFault`	IPC message was not transmitted due to the IPC message source.
`rcTargetFault`	IPC message was not transmitted due to the IPC message recipient.
`rcIpcInterrupt`	IPC was interrupted by another process thread.
`rcRestart`	Indicates that the function needs to be called again.
`rcFail`	The function ended with an error.
`rcNoCapability`	The operation cannot be performed on the resource.
`rcNotReady`	Initialization failed.
`rcUnimplemented`	The function was not implemented.
`rcBufferTooLarge`	Large buffer size.
`rcBusy`	Resource temporarily unavailable.
`rcResourceNotFound`	Resource not found.
`rcTimeout`	Timed out.
`rcSecurityDisallow`	The operation was denied by security mechanisms.
`rcFutexWouldBlock`	The operation will result in a block.
`rcAbort`	The operation was aborted.
`rcInvalidThreadState`	Invalid function called in the interrupt handler.
`rcAlreadyExists`	Set of elements already contains the element being added.
`rcInvalidOperation`	Operation cannot be completed.
`rcHandleRevoked`	Resource access rights were revoked.
`rcQuotaExceeded`	Resource quota exceeded.

`rcDeviceNotFound`	Device not found.
`rcOverflow`	An overflow occurred.
`rcAlreadyDone`	Operation has already been completed.
`rcNotRun`	Operation has not been started.

Defining error codes

To define an error code, the developer of a KasperskyOS-based solution needs to use the `MAKE_RETCODE()` macro defined in the `sysroot-*-kos/include/rtl/retcode.h` header file from the KasperskyOS SDK. The developer must also use the `customer` parameter to pass the symbolic constant `RC_CUSTOMER_TRUE`.

Example:

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

> An error description that is passed via the `desc` parameter is not used by the `MAKE_RETCODE()` macro. This description is needed to create a database of error codes when building a KasperskyOS-based solution. At present, a mechanism for creating and using such a database has not been implemented.

Reading error code structure fields

The `RC_GET_CUSTOMER()`, `RC_GET_SPACE()`, `RC_GET_FACILITY()` and `RC_GET_CODE()` macros defined in the `sysroot-*-kos/include/rtl/retcode.h` header file from the KasperskyOS SDK let you read error code structure fields.

The `RETCODE_HR_PARAMS()` and `RETCODE_HR_FMT()` macros defined in the `sysroot-*-kos/include/rtl/retcode_hr.h` header file from the KasperskyOS SDK are used for formatted display of error details.

libkos library

The `libkos` library is the basic KasperskyOS library that provides the set of APIs that allow programs and other libraries (for example, `libc` and `kdf`) to use core endpoints. The APIs provided by the `libkos` library enable solution developers to do the following:

- Manage processes, threads, and virtual memory.
- Control access to resources.
- Perform input/output operations.
- Create IPC channels.
- Manage power.
- Obtain statistical data on the system.

- Use other capabilities supported by core endpoints.

> On hardware platforms running an Arm processor architecture, input and output parameters of the `libkos` library API cannot be saved in "Device memory" because this could lead to unpredictable behavior. (Exceptions to this rule are the `addr` parameter of the `KnVmQuery()` function from the vmm_api.h API, the `reg` and `baseReg` parameters of functions from the mmio.h API, the `va` parameter of the `KnHalFlushCache()` function from the hal_api.h API, and the `va` parameter of the `KosCpuCacheFlush()` function from the cpucache.h API.) Parameters of the `libkos` library API must be saved in "Normal memory". To copy data from "Device memory" to "Normal memory" and vice versa, you must use the `RtlPedanticMemcpy()` function declared in the header file `sysroot-*-kos/include/rtl/string_pedantic.h` from the KasperskyOS SDK.

Managing handles (handle_api.h)

The API is defined in the `sysroot-*-kos/include/coresrv/handle/handle_api.h` header file from the KasperskyOS SDK.

The API is intended for performing operations with handles. Handles have the `Handle` type, which is defined in the header file `sysroot-*-kos/include/handle/handletype.h` from the KasperskyOS SDK.

Locality of handles

Each process receives handles from its own handle space irrespective of other processes. The handle spaces of different processes are absolutely identical in that they consist of the same set of values. Therefore, a handle is unique (has a unique value) only within the handle space of the single process that owns the particular handle. In other words, different processes may have identical handles that identify different resources, or may have different handles that identify the same resource.

Handle permissions mask

A handle permissions mask has a size of 32 bits and consists of a general part and a specialized part. The general part describes the general rights that are not specific to any particular resource (the flags of these rights are defined in the header file `sysroot-*-kos/include/services/ocap.h` from the KasperskyOS SDK). For example, the general part contains the `OCAP_HANDLE_TRANSFER` flag, which defines the permission to transfer the handle. The specialized part describes the rights that are specific to the particular user resource or system resource. The flags of the specialized part's permissions for system resources are defined in the `ocap.h` header file. The structure of the specialized part for user resources is defined by the resource provider by using the `OCAP_HANDLE_SPEC()` macro that is defined in the `ocap.h` header file. The resource provider must export the public header files describing the flags of the specialized part.

When the handle of a system resource is created, the permissions mask is defined by the KasperskyOS kernel, which applies permissions masks from the `ocap.h` header file. It applies permissions masks with names such as `OCAP_*_FULL` (for example, `OCAP_IOPORT_FULL`, `OCAP_TASK_FULL`, `OCAP_FILE_FULL`) and `OCAP_IPC_*` (for example, `OCAP_IPC_SERVER`, `OCAP_IPC_LISTENER`, `OCAP_IPC_CLIENT`).

When the handle of a user resource is created, the permissions mask is defined by the user.

When a handle is transferred, the permissions mask is defined by the user but the transferred access rights cannot be elevated above the access rights of the process.

Creating handles

Information about API functions is provided in the table below.

Creating handles of system resources

Handles of system resources are created when these resources are created. For example, handles are created when an interrupt or MMIO memory region is registered, and when a DMA buffer, thread, or process is created.

Creating handles of user resources

Handles of user resources are created by the providers of these resources by using the `KnHandleCreateUserObject()` or `KnHandleCreateUserObjectEx()` function.

The context of a user resource must be defined through the `context` parameter. The *user resource context* consists of data that allows the resource provider to identify the resource and its state when access to the resource is requested by other processes. This normally consists of a data set with various types of data (structure). For example, the context of a file may include the name, path, and cursor position. The user resource context is used as the resource transfer context or is used together with multiple resource transfer contexts.

You must use the `rights` parameter to define the handle permissions mask.

Creating IPC handles

An *IPC handle* is a handle that identifies an IPC channel. IPC handles are used to execute system calls. A client IPC handle is necessary for executing a `Call()` system call. A server IPC handle is necessary for executing the `Recv()` and `Reply()` system calls. A *listener handle* is a server IPC handle that has extended rights allowing it to add IPC channels to the set of IPC channels identified by this handle. A *callable handle* is a client IPC handle that simultaneously identifies the IPC channel to a server and an endpoint of this server.

A server creates a callable handle and passes it to a client so that the client can use the server endpoint. The client initializes IPC transport by using the callable handle that it received. In addition, the client specifies the `INVALID_RIID` value as the Runtime Implementation Identifier (RIID) in the proxy object initialization function. To create a callable handle, call the `KnHandleCreateUserObjectEx()` function and specify the server IPC handle and the RIID in the `ipcChannel` and `riid` parameters, respectively. Use the `context` parameter to specify the data to be associated with the callable handle. The server will be able to receive the pointer to this data when dereferencing the callable handle. (Even though the callable handle is an IPC handle, the kernel puts it into the `base_.self` field of the constant part of an IPC request.)

To create the client IPC handle, server IPC handle, and listener IPC handle and associate them with each other, call the `KnHandleConnect()` or `KnHandleConnectEx()` function. These functions are used to statically create IPC channels. The `KnHandleConnect()` function creates IPC handles from the handle space of the calling process. However, the client IPC handle can be transferred to another process. The `KnHandleConnectEx()` function can create IPC handles from the handle space of the calling process or from the handle spaces of other processes, such as the client and server.

When calling the `KnHandleConnect()` or `KnHandleConnectEx()` function with the `INVALID_HANDLE` value in the parameter that defines the listener handle, a new listener handle is created. However, the server IPC handle and listener IPC handle in the output parameters are the same handle. If a listener handle is specified when calling the `KnHandleConnect()` or `KnHandleConnectEx()` function, the created server IPC handle will provide the capability to receive IPC requests over all IPC channels associated with this listener handle. In this case, the server IPC handle and listener IPC handle in the output parameters are different handles. (The first IPC channel associated with the listener handle is created when calling the `KnHandleConnect()` or `KnHandleConnectEx()` function with the `INVALID_HANDLE` value in the parameter that defines the listener handle. The second and subsequent IPC channels associated with the listener handle are created during the second and subsequent calls of the `KnHandleConnect()` or `KnHandleConnectEx()` function specifying the listener handle that was obtained during the first call.)

To call a listener handle that is not associated with a client IPC handle and server IPC handle, call the `KnHandleCreateListener()` function. (The `KnHandleConnect()` and `KnHandleConnectEx()` functions create a listener handle associated with a client IPC handle and server IPC handle.) The `KnHandleCreateListener()` function is convenient for creating a listener handle that will be subsequently bound to callable handles.

To create a client IPC handle for querying the Kaspersky Security Module through the security interface, call the `KnHandleSecurityConnect()` function. This function is called by the `libkos` library when initializing IPC transport for querying the security module.

Information about API functions

handle_api.h functions

Function	Information about the function
`KnHandleCreateUserObject()`	Purpose Creates a handle. Parameters • [in] `type` – handle type. Fictitious parameter that must take a value ranging from the `HANDLE_TYPE_USER_FIRST` constant to the `HANDLE_TYPE_USER_LAST` constant as defined in the header file `sysroot-*-kos/include/handle/handletype.h` from the KasperskyOS SDK. • [in] `rights` – handle permissions mask. • [in,optional] `context` – pointer to the data that should be associated with the handle, or `RTL_NULL` if this association is not required. • [out] `handle` – pointer to the handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnHandleCreateUserObjectEx()`	Purpose Creates a handle. Parameters • [in] `type` – handle type. Fictitious parameter that must take a value ranging from the `HANDLE_TYPE_USER_FIRST` constant to the `HANDLE_TYPE_USER_LAST` constant as defined in the header file `sysroot-*-kos/include/handle/handletype.h` from the KasperskyOS SDK. • [in] `rights` – handle permissions mask. • [in,optional] `context` – pointer to the data that should be associated with the handle, or `RTL_NULL` if this association is not required. • [in,optional] `ipcChannel` – server IPC handle, or `INVALID_HANDLE` if you do not need to create a callable handle.

	• [in,optional] `riid` – Runtime Implementation Identifier (RIID), or `INVALID_RIID` if you do not need to create a callable handle. • [out] `handle` – pointer to the handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnHandleConnect()`	Purpose Creates and connects the client, server, and listener IPC handles. Parameters • [in,optional] `ls` – listener handle, or `INVALID_HANDLE` if you need to create it. • [out,optional] `outLs` – pointer to the listener handle. You can specify `RTL_NULL` if the `ls` parameter is used to define the listener handle. • [out,optional] `outSr` – pointer to the server IPC handle, or `RTL_NULL` to not create a server IPC handle if the `ls` parameter is used to define the listener handle. • [out] `outCl` – pointer to the client IPC handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnHandleConnectEx()`	Purpose Creates and connects the client, server, and listener IPC handles. Parameters • [in] `server` – handle of the server process. • [in,optional] `srListener` – listener handle from the handle space of the server process, or `INVALID_HANDLE` if you need to create it. • [in] `client` – handle of the client process. • [out,optional] `outSrListener` – pointer to the listener handle from the handle space of the server process. You can specify `RTL_NULL` if the `srListener` parameter is used to define the listener handle. • [out,optional] `outSrEndpoint` – pointer to the server IPC handle from the handle space of the server process, or `RTL_NULL` to not create a server IPC handle if the `srListener` parameter is used to define the listener handle. • [out] `outClEndpoint` – pointer to the client IPC handle from the handle space of the client process. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnHandleSecurityConnect()`	Purpose Creates a client IPC handle for querying the Kaspersky Security Module through the security interface. Parameters • [out] `client` – pointer to the handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnHandleCreateListener()`	Purpose Creates a listener handle that is not associated with a client IPC handle and server IPC handle. Parameters

	• [out] `listener` – pointer to the listener handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Transferring handles

Information about API functions is provided in the table below.

Overview of handle transfers

Handles are transferred between processes so that resource consumers can gain access to required resources. Due to the specific locality of handles, a handle transfer initiates the creation of a handle from the handle space of the recipient process. This handle is registered as a descendant of the transferred handle and identifies the same resource.

One handle can be transferred multiple times to one or more processes. Each transfer initiates the creation of a new descendant of the transferred handle on the recipient process side. A process can transfer handles that it received from other processes or the KasperskyOS kernel. For this reason, a handle may have multiple generations of descendants. The generation hierarchy of handles for each resource is stored in the KasperskyOS kernel in the form of a *handle inheritance tree*.

A process can transfer handles for user resources and system resources if the access rights of these handles permit such a transfer (the `OCAP_HANDLE_TRANSFER` flag is set in the permissions mask). A descendant may have less access rights than an ancestor. For example, a transferring process with read-and-write permissions for a file can transfer read-only permissions. The transferring process can also prohibit the recipient process from further transferring the handle. Access rights are defined in the transferred permissions mask for the handle.

Conditions for transferring handles

To enable processes to transfer handles to other processes, the following conditions must be met:

1. An IPC channel is created between the processes.
2. The solution security policy (`security.psl`) allows interaction between process classes.
3. Interface methods are implemented for transferring handles.

> The API task.h enables a parent process to pass handles to a child process that is not yet running.

In an IDL description, signatures of interface methods for transferring handles have input (`in`) and/or output (`out`) parameters of the `Handle` type or `array` type with elements of the `Handle` type. Up to 255 handles can be passed through the input parameters of one method. This same number of handles can be received through output parameters.

Example IDL description that defines the signatures of interface methods for transferring handles:

```
package IpcTransfer
interface {
```

```
    PublishResource1(in Handle handle, out UInt32 result);
    PublishResource7(in Handle handle1, in Handle handle2,
                     in Handle handle3, in Handle handle4,
                     in Handle handle5, in Handle handle6,
                     in Handle handle7, out UInt32 result);
    OpenResource(in UInt32 ID, out Handle handle);
}
```

For each parameter of the `Handle` type, the NK compiler generates a field of the `nk_handle_desc_t` type (hereinafter also referred to as the *transport container of the handle*) in the `*_req` IPC request structure and/or `*_res` IPC response structure. This type is declared in the header file `sysroot-*-kos/include/nk/types.h` from the KasperskyOS SDK and comprises a structure consisting of the following three fields: `handle` field for the handle, `rights` field for the handle permissions mask, and the `badge` field for the resource transfer context.

Resource transfer context

The *resource transfer context* consists of data that allows the server to identify the resource and its state when access to the resource is requested via descendants of the transferred handle. This normally consists of a data set with various types of data (structure). For example, the transfer context of a file may include the name, path, and cursor position. The server receives a pointer to the resource transfer context when dereferencing a handle.

Regardless of whether or not the server is the resource provider, the server can associate each handle transfer with a separate resource transfer context. This resource transfer context is bound only to the handle descendants (handle inheritance subtree) that were generated as a result of a specific transfer of the handle. This lets you define the state of a resource in relation to a separate transfer of the handle of this resource. For example, for cases when one file may be accessed multiple times, the file transfer context lets you define which specific opening of this file corresponds to a received IPC request.

If the server is the resource provider, each transfer of the handle of this resource is associated with the user resource context by default. In other words, the user resource context is used as the resource transfer context for each handle transfer if the particular transfer is not associated with a separate resource transfer context.

A server that is the resource provider can use both the user resource context and the resource transfer context together. For example, the name, path and size of a file is stored in the user resource context while the cursor position can be stored in multiple resource transfer contexts because each client can work with different parts of the file. Technically, joint use of the user resource context and resource transfer contexts is possible because the resource transfer contexts store a pointer to the user resource context.

If the client uses multiple various-type resources of the server, the resource transfer contexts (or contexts of user resources if they are used as resource transfer contexts) must be typified objects of the `KosObject` type (for details about `KosObjects`, see Using KosObjects (objects.h)). This is necessary so that the server can verify that the client using a resource has sent the interface method the handle of the specific resource that corresponds to this method. This verification is required because the client could mistakenly send the interface method a resource handle that does not correspond to this method. For example, a client may have received a file handle and sent it to an interface method for working with volumes.

To associate a handle transfer with a resource transfer context, the server puts the handle of the resource transfer context object into the `badge` field of the `nk_handle_desc_t` structure. The *resource transfer context object* is the kernel object that stores the pointer to the resource transfer context. To create a resource transfer context object, call the `KnHandleCreateBadge()` function. This function is bound to the notification mechanism because a server needs to know when a resource transfer context object will be closed and deleted. The server needs this information to free up or re-use memory that was allotted for storing the resource transfer context.

The resource transfer context object will be closed upon the closure or revocation of the handle descendants that comprise the handle inheritance subtree whose root node was generated by the transfer of this handle in association with this object. (A transferred handle may be closed intentionally or unintentionally, such as when a recipient client is unexpectedly terminated.) After receiving a notification regarding the closure of a resource transfer context object, the server closes the handle of this object. After this, the resource transfer context object will be deleted. After receiving a notification regarding the deletion of the resource transfer context object, the server frees up or re-uses the memory that was allotted for storing the resource transfer context.

One resource transfer context object can be associated with only one handle transfer.

Packaging data into the transport container of a handle

To package a handle, handle permissions mask, and resource transfer context object handle into a handle transport container, use the `nk_handle_desc()` macro that is defined in the header file `sysroot-*-kos/include/nk/types.h` from the KasperskyOS SDK. This macro receives a variable number of parameters.

If no parameter is passed to the macro, the `NK_INVALID_HANDLE` value will be written to the `handle` field of the `nk_handle_desc_t` structure. If one parameter is passed to the macro, this parameter is interpreted as the handle. If two parameters are passed to the macro, the first parameter is interpreted as the handle and the second parameter is interpreted as the handle permissions mask. If three parameters are passed to the macro, the first parameter is interpreted as the handle, the second parameter is interpreted as the handle permissions mask, and the third parameter is interpreted as the resource transfer context object handle.

Extracting data from the transport container of a handle

To extract the handle, handle permissions mask, and pointer to the resource transfer context from the transport container of a handle, use the `nk_get_handle()`, `nk_get_rights()` and `nk_get_badge_op()` (or `nk_get_badge()`) functions, respectively, which are declared in the header file `sysroot-*-kos/include/nk/types.h` from the KasperskyOS SDK. The `nk_get_badge_op()` and `nk_get_badge()` functions should be used only when dereferencing handles.

Handle transfer scenarios

The scenario for transferring handles from a client to the server includes the following steps:

1. The client packages the handles and handle permissions masks into fields of the `*_req` IPC requests structure of the `nk_handle_desc_t` type.

2. The client calls the interface method for transferring handles to the server. The `Call()` system call is executed when this method is called.

3. The server receives an IPC request by executing the `Recv()` system call.

4. The dispatcher on the server side calls the method corresponding to the IPC request. This method extracts the handles and handle permissions masks from fields of the `*_req` IPC request structure of the `nk_handle_desc_t` type.

The scenario for transferring handles from the server to a client includes the following steps:

1. The client calls the interface method for receiving handles from the server. The `Call()` system call is executed when this method is called.

2. The server receives an IPC request by executing the `Recv()` system call.

3. The dispatcher on the server side calls the method corresponding to the IPC request. This method packages the handles, handle permissions masks and resource transfer context object handles into fields of the `*_res` IPC response structure of the `nk_handle_desc_t` type.

4. The server responds to the IPC request by executing the `Reply()` system call.

5. On the client side, the interface method returns control. After this, the client extracts the handles and handle permissions masks from fields of the `*_res` IPC response structure of the `nk_handle_desc_t` type.

If the transferring process defines more access rights in the transferred handle permissions mask than the access rights defined for the transferred handle (which it owns), the transfer is not completed. In this case, the `Call()` system call executed by the transferring or recipient client or the `Reply()` system call executed by the transferring server ends with the `rcSecurityDisallow` error.

Information about API functions

handle_api.h functions

Function	Information about the function
`KnHandleCreateBadge()`	Purpose Creates a resource transfer context object and configures a notification mechanism for monitoring the life cycle of this object. Parameters • [in] `notice` – identifier of the notification receiver. • [in] `eventId` – identifier of the "resource–event mask" entry in the notification receiver. • [in,optional] `context` – pointer to the data that should be associated with the handle transfer, or `RTL_NULL` if this association is not required. • [out] `handle` – pointer to the handle of the resource transfer context object. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information The notification receiver is configured to receive notifications about events that match the `EVENT_OBJECT_DESTROYED` and `EVENT_BADGE_CLOSED` flags of the event mask.

Duplicating handles

Handle duplication is similar to a handle transfer, but duplication is performed within a process. A handle descendant is created in the same process and from the same handle space. The rights of the handle descendant may be less than or equal to the rights of the original handle. Handle duplication can be associated with a resource transfer context object. This lets you use the notification mechanism to track the closure or revocation of all handle descendants that form the handle inheritance subtree whose root node was generated by the duplication operation. It also provides the capability to revoke these descendants.

To duplicate a handle, call the `KnHandleCopy()` function. To do so, the `OCAP_HANDLE_COPY` flag must be set in the handle permissions mask.

Information about API functions is provided in the table below.

handle_api.h functions

Function	Information about the function
`KnHandleCopy()`	Purpose Duplicates a handle. As a result of duplication, the calling process receives the handle descendant. Parameters • [in] `inHandle` – original handle. • [in] `newRightsMask` – permissions mask of the handle descendant. • [in,optional] `copyBadge` – handle of the resource transfer context object, or `INVALID_HANDLE` if you do not need to associate handle duplication with this object. • [out] `outHandle` – pointer to the handle descendant. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Dereferencing handles

When *dereferencing a handle*, the client sends the handle to the server, and the server receives a pointer to the resource transfer context, the permissions mask of the sent handle, and the ancestor of the handle sent by the client and already owned by the server. Dereferencing occurs when a resource consumer that called methods for working with a resource (such as read/write or access closure) sends the resource provider the handle that was received from this resource provider when access to the resource was opened.

Dereferencing handles requires fulfillment of the same conditions and utilizes the same mechanisms and data types as when transferring handles. A handle dereferencing scenario includes the following steps:

1. The client packages the handle into a field of the `*_req` IPC request structure of the `nk_handle_desc_t` type.
2. The client calls the interface method for sending the handle to the server for the purpose of performing operations with the resource. The `Call()` system call is executed when this method is called.
3. The server receives the IPC request by executing the `Recv()` system call.
4. The dispatcher on the server side calls the method corresponding to the IPC request. This method verifies that the dereferencing operation was specifically executed instead of a handle transfer. Then the called method has the option to verify that the access rights of the dereferenced handle (that was sent by the client) permit the requested actions with the resource, and extracts the pointer to the resource transfer context from the field of the `*_req` request structure of the `nk_handle_desc_t` type.

To perform verification, the server uses the `nk_is_handle_dereferenced()` and `nk_get_badge_op()` functions that are declared in the header file `sysroot-*-kos/include/nk/types.h` from the KasperskyOS SDK.

types.h (fragment)

```
/**
* Returns a value different from null if
* the handle in the transport container of
* "desc" is received as a result of dereferencing
* the handle.Returns null if the handle
* in the transport container of "desc" is received
* as a result of a handle transfer.
*/
```

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

/**
* Extracts the pointer to the resource transfer context
* "badge" from the transport container of "desc"
* if the permissions mask that was put in the transport
* container of the desc handle has the operation flags set.
* If successful, the function returns NK_EOK, otherwise it returns an error code.
*/
static inline
nk_err_t nk_get_badge_op(const nk_handle_desc_t *desc,
                         nk_rights_t operation,
                         nk_badge_t *badge)
```

Generally, the server does not require the handle that was received from dereferencing because the server normally retains the handles that it owns, for example, within the contexts of user resources. However, the server can extract this handle from the handle transport container if necessary.

Revoking handles

A process can revoke descendants of a handle that it owns. Handles are revoked according to the handle inheritance tree.

Revoked handles are not closed. However, you cannot query resources via revoked handles. Any function that receives the handle will end with the `rcHandleRevoked` error if the function is called with a revoked handle.

To revoke handle descendants, call the `KnHandleRevoke()` or `KnHandleRevokeSubtree()` function. The `KnHandleRevokeSubtree()` function uses the resource transfer context object that is created when transferring handles.

If each handle of a system resource in all processes that own these handles are closed (see Closing handles) or revoked, this system resource will be deleted.

Information about API functions is provided in the table below.

handle_api.h functions

Function	Information about the function
`KnHandleRevoke()`	Purpose Closes a handle and revokes its descendants. Parameters • [in] `handle` – a handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnHandleRevokeSubtree()`	Purpose Revokes the handles that make up the inheritance subtree of the specified handle. Parameters • [in] `handle` – a handle. The handles forming the inheritance subtree of this handle are revoked. • [in] `badge` – handle that identifies the resource transfer context object, which defines the inheritance subtree of the handles to revoke. The root node of this subtree is the handle that was generated by the transfer or duplication of the handle that is defined through the `handle` parameter and is associated with the resource transfer context object. Returned values

	If successful, the function returns `rcOk`, otherwise it returns an error code.

Closing handles

A process can close the handles that it owns. Closing a handle terminates the association between an ID and a resource, thereby releasing the ID. Closing a handle does not invalidate its ancestors and descendants (in contrast to revoking a handle, which actually invalidates the descendants of the handle). In other words, the ancestors and descendants of a closed handle can still be used to provide access to the resource that they identify. Also, closing a handle does not disrupt the handle inheritance tree associated with the resource identified by the particular handle. The place of a closed handle is occupied by its ancestor. In other words, the ancestor of a closed handle becomes the direct ancestor of the descendants of the closed handle.

To close the handle, call the `KnHandleClose()` function.

If each handle of a system resource in all processes that own these handles are revoked (see Revoking handles) or closed, this system resource will be deleted.

Information about API functions is provided in the table below.

handle_api.h functions

Function	Information about the function
`KnHandleClose()`	Purpose Closes a handle. Parameters • [in] `handle` – a handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Getting a security ID (SID)

By getting the SID values for different handles, you can determine whether these handles identify different resources or the same resource.

To get a security ID for a handle, call the `KnHandleGetSidByHandle()` function. To do so, the `OCAP_HANDLE_GET_SID` flag must be set in the handle permissions mask.

Information about API functions is provided in the table below.

handle_api.h functions

Function	Information about the function
`KnHandleGetSidByHandle()`	Purpose Receives a security ID (SID) based on a handle. Parameters • [in] `handle` – a handle. • [out] `sid` – pointer to the security ID (SID). Returned values

	If successful, the function returns `rcOk`, otherwise it returns an error code.

OCap usage example

This example describes an OCap usage scenario in which the resource provider provides the following methods for accessing its resources:

- `OpenResource()` – opens access to the resource.
- `UseResource()` – uses the resource.
- `CloseResource()` – closes access to the resource.

The resource consumer uses these methods.

IDL description:

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

The scenario includes the following steps:

1. The resource provider creates the user resource context and calls the `KnHandleCreateUserObject()` function to create the resource handle. The resource provider saves the resource handle in the user resource context.
2. The resource consumer calls the `OpenResource()` method to open access to the resource.
   a. The resource provider creates the resource transfer context and calls the `KnHandleCreateBadge()` function to create a resource transfer context object and configure the notification receiver to receive notifications regarding the closure and deletion of the resource transfer context object. The resource provider saves the handle of the resource transfer context object and the pointer to the user resource context in the resource transfer context.
   b. The resource provider uses the `nk_handle_desc()` macro to package the resource handle, permissions mask of the handle, and pointer to the resource transfer context object into the handle transport container.
   c. The handle is transferred from the resource provider to the resource consumer, which means that the resource consumer receives a descendant of the handle owned by the resource provider.
   d. The `OpenResource()` method call completes successfully. The resource consumer extracts the handle and permissions mask of the handle from the handle transport container by using the `nk_get_handle()` and `nk_get_rights()` functions, respectively. The handle permissions mask is not required by the resource consumer to query the resource, but is transferred so that the resource consumer can find out its permissions for accessing the resource.
3. The resource consumer calls the `UseResource()` method to utilize the resource.
   a. The handle that was received from the resource provider at step 2 is used as a parameter of the `UseResource()` method. Before calling this method, the resource consumer uses the `nk_handle_desc()` macro to package the handle into the handle transport container.

b. The handle is dereferenced, after which the resource provider receives the pointer to the resource transfer context.

c. The resource provider uses the `nk_is_handle_dereferenced()` function to verify that the dereferencing operation was completed instead of a handle transfer.

d. The resource provider verifies that the access rights of the dereferenced handle (that was sent by the resource consumer) allows the requested operation with the resource, and extracts the pointer to the resource transfer context from the handle transport container. To do so, the resource provider uses the `nk_get_badge_op()` function, which extracts the pointer to the resource transfer context from the handle transport container if the received permissions mask has the corresponding flags set for the requested operation.

e. The resource provider uses the resource transfer context and the user resource context to perform the corresponding operation with the resource as requested by the resource consumer. Then the resource provider sends the results of this operation to the resource consumer.

f. The `UseResource()` method call completes successfully. The resource consumer receives the results of the operation performed with the resource.

4. The resource consumer calls the `CloseResource()` method to close access to the resource.

   a. The handle that was received from the resource provider at step 2 is used as a parameter of the `CloseResource()` method. Before calling this method, the resource consumer uses the `nk_handle_desc()` macro to package the handle into the handle transport container. After the `CloseResource()` method is called, the resource consumer uses the `KnHandleClose()` function to close the handle.

   b. The handle is dereferenced, after which the resource provider receives the pointer to the resource transfer context.

   c. The resource provider uses the `nk_is_handle_dereferenced()` function to verify that the dereferencing operation was completed instead of a handle transfer.

   d. The resource provider uses the `nk_get_badge()` function to extract the pointer to the resource transfer context from the handle transport container.

   e. The resource provider uses the `KnHandleRevokeSubtree()` function to revoke the handle owned by the resource consumer. The resource handle owned by the resource provider and the handle of the resource transfer context object are used as parameters of this function. The resource provider obtains access to these handles through the pointer to the resource transfer context. (Technically, the handle owned by the resource consumer does not have to be revoked because the resource consumer already closed it. However, the revoke operation is performed in case the resource provider is not sure if the resource consumer actually closed the handle).

   f. The `CloseResource()` method call completes successfully.

5. The resource provider frees up the memory that was allocated for the resource transfer context and user resource context.

   a. The resource provider calls the `KnNoticeGetEvent()` function to receive a notification that the resource transfer context object was closed, and uses the `KnHandleClose()` function to close the handle of the resource transfer context object.

   b. The resource provider calls the `KnNoticeGetEvent()` function to receive a notification that the resource transfer context object was deleted, and frees up the memory that was allocated for the resource transfer context.

c. The resource provider uses the `KnHandleClose()` function to close the resource handle and free up the memory that was allocated for the user resource context.

Managing virtual memory (vmm_api.h)

The API is defined in the header file `sysroot-*-kos/include/coresrv/vmm/vmm_api.h` from the KasperskyOS SDK.

The API is designed to allocate and free memory, create shared memory, and prepare ELF image segments to be loaded into process memory.

Allocating and freeing memory

Information about API functions is provided in the table below.

Using the API

Virtual memory pages may be free, reserved, or committed. Free pages that are used to allocate a virtual memory region become reserved pages. Reserved pages may or may not be mapped to physical memory. Reserved pages that are mapped to physical memory are committed pages.

Pages of a virtual memory region that are allocated by the `KnVmAllocate()` function call can be committed in three different ways:

1. Fully committed when the region is allocated.

2. Fully or partially committed after allocation of the region (by calling the `KnVmCommit()` function).

3. Committed whenever virtual addresses are queried (so-called "lazy" mode).

When the first option is used, the entire required volume of physical memory is allocated immediately after the virtual memory region is reserved. When the second or third option is used, the virtual memory region is reserved without allocating physical memory. This lets you conserve physical memory if a reserved virtual memory region will not actually be required or will only be partially used. In addition, a virtual memory region can be allocated faster if its allocation does not include commitment.

When in "lazy" mode, physical memory is allocated only when the virtual memory region is actually queried. In this case, the page containing the queried address and several pages before and after this address are committed.

If you call the `KnVmAllocate()` function with the `VMM_FLAG_COMMIT` and `VMM_FLAG_LOCKED` flags, the virtual memory region will be reserved and fully committed. If you call the `KnVmAllocate()` function with the `VMM_FLAG_COMMIT` flag but without the `VMM_FLAG_LOCKED` flag, the virtual memory region will be reserved and committed only in "lazy" mode. If you call the `KnVmAllocate()` function with the `VMM_FLAG_LOCKED` flag but without the `VMM_FLAG_COMMIT` flag, the virtual memory region will be reserved without commitment and a subsequent call of the `KnVmCommit()` function will fully commit this region. If you call the `KnVmAllocate()` function without the `VMM_FLAG_COMMIT` and `VMM_FLAG_LOCKED` flags, the virtual memory region will be reserved without commitment and a subsequent call of the `KnVmCommit()` function will commit this region in "lazy" mode.

A virtual memory region is allocated when the MDL buffer, DMA buffer or MMIO memory region is mapped to process memory. This region is allocated by the mapping function.

A protected page may reside at the beginning and/or end of the virtual memory region. This page is never committed. Any query of this page will result in an exception signaling that the region boundaries have been exceeded.

To change the access rights to the virtual memory region, call the `KnVmProtect()` function. You can fully close and then re-open access to a region while retaining is contents.

To free physical memory while retaining the reservation of virtual addresses, call the `KnVmDecommit()` or `KnVmReset()` function. In this case, the contents of the virtual memory region will be lost. After freeing physical memory by calling the `KnVmDecommit()` function, you must call the `KnVmCommit()` function to subsequently use the virtual memory region. After freeing physical memory by calling the `KnVmReset()` function, the virtual memory region can be used without any additional actions required. This virtual memory region will correspond to memory allocated by calling the `KnVmAllocate()` function with the `VMM_FLAG_COMMIT` flag but without the `VMM_FLAG_LOCKED` flag.

> The `KnVmProtect()`, `KnVmDecommit()`, and `KnVmReset()` functions cannot be used if the MDL buffer, DMA buffer or MMIO memory region is mapped to the virtual memory region.

To free the virtual memory region, call the `KnVmUnmap()` function. When this function is called, the reserved pages will become free regardless of whether or not they were committed, and the physical memory mapped to committed pages is also freed.

> The `KnVmUnmap()` function frees the virtual addresses of a region mapped to the MDL buffer, but does not delete the MDL buffer. In addition, this function cannot be used if the DMA buffer or MMIO memory region is mapped to the virtual memory region.

The `KnVmCommit()`, `KnVmProtect()`, `KnVmDecommit()`, `KnVmReset()`, and `KnVmUnmap()` functions can be applied for the entire allocated virtual memory region or for a portion of it.

The functions presented in this section provide the basis for implementing memory allocation/deallocation functions of the libkos library, and functions of POSIX interfaces such as `malloc()`, `calloc()`, `realloc()`, `free()`, `mmap()`, and `munmap()`.

Information about API functions

vmm_api.h functions

Function	Information about the function
`KnVmAllocate()`	Purpose Allocates (reserves and optionally commits) a virtual memory region. Parameters • [in,optional] `addr` – page-aligned, preferred base address of the virtual memory region, or `0` to select this address automatically. • [in] `size` – size of the virtual memory region in bytes. It must be a multiple of the memory page size. • [in] `flags` – flags defining the parameters of the virtual memory region. The flags are defined in the header file `sysroot-*-kos/include/vmm/flags.h` from the KasperskyOS SDK. Returned values

	If successful, the function returns the base address of the allocated virtual memory region, otherwise it returns `RTL_NULL`. Additional information In the `flags` parameter, you can specify the following flags: • `VMM_FLAG_COMMIT` – commits the virtual memory region in "lazy" mode, whereby the physical memory pages are allocated as virtual addresses are queried. • `VMM_FLAG_LOCKED` – commits the virtual memory region while allocating the entire required volume of physical memory. • `VMM_FLAG_READ`, `VMM_FLAG_WRITE`, `VMM_FLAG_EXECUTE` and `VMM_FLAG_RWX_MASK` – flags defining the access rights to the virtual memory region. • `VMM_FLAG_LOW_GUARD`, `VMM_FLAG_HIGH_GUARD` – adds a protected page to the beginning and end of the virtual memory region, respectively. The size of the protected page is not included in the size of the virtual memory region. • `VMM_FLAG_GROW_DOWN` – defines the direction in which the virtual memory is mapped to physical memory when "lazy" commitment mode is applied. If this flag is set, the virtual memory page containing the queried address and several preceding pages are mapped to physical memory. If this flag is not set, the virtual memory page containing the queried address and several subsequent pages are mapped to physical memory. Permissible combinations of flags defining the access rights to the virtual memory region: • `VMM_FLAG_READ` – read access. • `VMM_FLAG_READ | VMM_FLAG_WRITE` – read-and-write access. • `VMM_FLAG_READ | VMM_FLAG_EXECUTE` – read-and-execute access. • `VMM_FLAG_RWX_MASK` or `VMM_FLAG_READ | VMM_FLAG_WRITE | VMM_FLAG_EXECUTE` – read/write/execute access.
`KnVmCommit()`	Purpose Commits a virtual memory region. Parameters • [in] `addr` – page-aligned base address of the virtual memory region. • [in] `size` – size of the virtual memory region in bytes. It must be a multiple of the memory page size. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnVmDecommit()`	Purpose Decommits a virtual memory region. Parameters • [in] `addr` – page-aligned base address of the virtual memory region. • [in] `size` – size of the virtual memory region in bytes. It must be a multiple of the memory page size. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnVmProtect()`	Purpose Modifies the access rights to the virtual memory region. Parameters • [in] `addr` – page-aligned base address of the virtual memory region. • [in] `size` – size of the virtual memory region in bytes. It must be a multiple of the memory page size.

	• [in] `newFlags` – flags defining the access rights to the virtual memory region. The flags are defined in the header file `sysroot-*-kos/include/vmm/flags.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information In the `newFlags` parameter, you can specify the following combinations of flags: • `VMM_FLAG_READ` – read access. • `VMM_FLAG_READ | VMM_FLAG_WRITE` – read-and-write access. • `VMM_FLAG_READ | VMM_FLAG_EXECUTE` – read-and-execute access. • `VMM_FLAG_RWX_MASK` or `VMM_FLAG_READ | VMM_FLAG_WRITE | VMM_FLAG_EXECUTE` – read/write/execute access. • `0` – access denied.
`KnVmUnmap()`	Purpose Frees up the virtual memory region. Parameters • [in] `addr` – page-aligned base address of the virtual memory region. • [in] `size` – size of the virtual memory region in bytes. It must be a multiple of the memory page size. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnVmQuery()`	Purpose Gets information about a virtual memory page. Parameters • [in] `addr` – address included in the virtual memory page. • [out] `info` – pointer to the structure containing information about the virtual memory page. The type of structure is defined in the header file `sysroot-*-kos/include/vmm/info.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnVmReset()`	Purpose Decommits a virtual memory region. Parameters • [in] `addr` – page-aligned base address of the virtual memory region. • [in] `size` – size of the virtual memory region in bytes. It must be a multiple of the memory page size. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Creating shared memory

Information about API functions is provided in the table below.

Using the API

An MDL buffer is used to create shared memory. An *MDL buffer* consists of one or more physical memory regions that can be mapped to the memory of multiple processes at the same time. A kernel object known as a memory descriptor list is used to map the MDL buffer to process memory. A *memory descriptor list* (MDL) is a data structure containing the addresses and sizes of physical memory regions comprising the MDL buffer. The MDL buffer handle identifies the memory descriptor list.

To create shared memory, a process needs to create an MDL buffer, map it to its own memory, and pass the MDL buffer handle via IPC to other processes that also need to map this MDL buffer to their own memory.

To create an MDL buffer, call the `KnPmmMdlCreate()`, `KnPmmMdlCreateFromBuf()` or `KnPmmMdlCreateFromVm()` function. The `KnPmmMdlCreateFromBuf()` function creates an MDL buffer and copies data to it. The `KnPmmMdlCreateFromVm()` function creates an MDL buffer and maps it to the memory of the calling process.

To reserve a virtual memory region and map the MDL buffer to it, call the `KnPmmMdlMap()` function. An MDL buffer can be mapped to multiple virtual memory regions of the same process.

An MDL buffer can be used to transfer large volumes of data between processes without creating shared memory. In this case, you must make sure that the MDL buffer is not mapped to the memory of multiple processes at the same time. Interacting processes must take turns mapping the MDL buffer to their own memory, reading and/or writing data and then freeing the virtual memory region that is mapped to this MDL buffer.

> The Kaspersky Security Module cannot control data that is transferred between processes via the MDL buffer.

Deleting an MDL buffer

To delete an MDL buffer, complete the following steps:

1. Free the virtual memory regions mapped to the MDL buffer in all processes that are using this MDL buffer.

   To complete this step, use the `KnVmUnmap()` function.

2. Close or revoke each MDL buffer handle in all processes that own these handles.

   To complete this step, use the `KnHandleClose()` and/or `KnHandleRevoke()` functions that are declared in the header file `sysroot-*-kos/include/coresrv/handle/handle_api.h` from the KasperskyOS SDK.

Information about API functions

vmm_api.h functions

Function	Information about the function
`KnPmmMdlCreate()`	Purpose Creates an MDL buffer. Parameters • [in] `size` – size of the MDL buffer in bytes. It must be a multiple of the memory page size. • [in] `flags` – flags defining the access rights to the MDL buffer. The parameter type and flags are defined in the header file `sysroot-*-kos/include/vmm/flags.h` from the KasperskyOS SDK. • [out] `outHandle` – pointer to the MDL buffer handle.

	Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information In the `flags` parameter, you can specify the following combinations of flags: • `VMM_FLAG_READ` – read access. • `VMM_FLAG_READ | VMM_FLAG_WRITE` – read-and-write access. • `VMM_FLAG_READ | VMM_FLAG_EXECUTE` – read-and-execute access. • `VMM_FLAG_RWX_MASK` or `VMM_FLAG_READ | VMM_FLAG_WRITE | VMM_FLAG_EXECUTE` – read/write/execute access.
`KnPmmMdlCreateFromVm()`	Purpose Creates an MDL buffer from physical memory that is mapped to the defined virtual memory region and maps the created MDL buffer to this region. Parameters • [in] `addr` – page-aligned base address of the virtual memory region. • [in] `size` – size of the virtual memory region in bytes. It must be a multiple of the memory page size. • [in] `flags` – flags defining the access rights to the MDL buffer. The parameter type and flags are defined in the header file `sysroot-*-kos/include/vmm/flags.h` from the KasperskyOS SDK. • [out] `outHandle` – pointer to the MDL buffer handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information This function can be used if the defined virtual memory region is allocated with commitment. In the `flags` parameter, you can specify the following combinations of flags: • `VMM_FLAG_READ` – read access. • `VMM_FLAG_READ | VMM_FLAG_WRITE` – read-and-write access. • `VMM_FLAG_READ | VMM_FLAG_EXECUTE` – read-and-execute access. • `VMM_FLAG_RWX_MASK` or `VMM_FLAG_READ | VMM_FLAG_WRITE | VMM_FLAG_EXECUTE` – read/write/execute access.
`KnPmmMdlCreateFromBuf()`	Purpose Creates an MDL buffer and copies data from the defined buffer to the MDL buffer. Parameters • [in] `offset` – offset (in bytes) in the MDL buffer where the data write operation must start. • [in] `size` – size of the MDL buffer in bytes. It must be a multiple of the memory page size. The following condition must also be fulfilled: *size>=bufSize+offset*. • [in] `flags` – flags defining the access rights to the MDL buffer. The parameter type and flags are defined in the header file `sysroot-*-kos/include/vmm/flags.h` from the KasperskyOS SDK. • [in] `buf` – pointer to the buffer containing the data. • [in] `bufSize` – size (in bytes) of the buffer containing the data. • [out] `outHandle` – pointer to the MDL buffer handle.

	Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information In the `flags` parameter, you can specify the following combinations of flags: • `VMM_FLAG_READ` – read access. • `VMM_FLAG_READ | VMM_FLAG_WRITE` – read-and-write access. • `VMM_FLAG_READ | VMM_FLAG_EXECUTE` – read-and-execute access. • `VMM_FLAG_RWX_MASK` or `VMM_FLAG_READ | VMM_FLAG_WRITE | VMM_FLAG_EXECUTE` – read/write/execute access.
`KnPmmMdlGetSize()`	Purpose Gets the size of the MDL buffer. Parameters • [in] `handle` – MDL buffer handle. • [out] `size` – size (in bytes) of the MDL buffer. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnPmmMdlMap()`	Purpose Reserves a virtual memory region and maps the MDL buffer to it. Parameters • [in] `handle` – MDL buffer handle. • [in] `offset` – offset (in bytes) in the MDL buffer where mapping should start. It must be a multiple of the memory page size. • [in] `length` – size (in bytes) of the part of the MDL buffer that needs to be mapped. It must be a multiple of the memory page size. The following condition must also be fulfilled: *length<=size of MDL buffer-offset*. • [in,optional] `hint` – page-aligned, preferred base address of the virtual memory region, or `0` to select this address automatically. • [in] `prots` – flags defining the parameters of the virtual memory region. The flags are defined in the header file `sysroot-*-kos/include/vmm/flags.h` from the KasperskyOS SDK. • [out] `addr` – pointer to the base address of the virtual memory region. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information In the `prots` parameter, you can specify the following flags: • `VMM_FLAG_READ`, `VMM_FLAG_WRITE`, `VMM_FLAG_EXECUTE` and `VMM_FLAG_RWX_MASK` – flags defining the access rights to the virtual memory region. • `VMM_FLAG_LOW_GUARD`, `VMM_FLAG_HIGH_GUARD` – adds a protected page to the beginning and end of the virtual memory region, respectively. Permissible combinations of flags defining the access rights to the virtual memory region: • `VMM_FLAG_READ` – read access. • `VMM_FLAG_READ | VMM_FLAG_WRITE` – read-and-write access.

	• `VMM_FLAG_READ` | `VMM_FLAG_EXECUTE` – read-and-execute access. • `VMM_FLAG_RWX_MASK` or `VMM_FLAG_READ` | `VMM_FLAG_WRITE` | `VMM_FLAG_EXECUTE` – read/write/execute access.
`KnPmmMdlClone()`	Purpose Creates an MDL buffer based on an existing one. The MDL buffer is created from the same regions of physical memory as the original buffer. Parameters • [in] `origin` – handle of the original MDL buffer. • [in] `offset` – offset (in bytes) in the original MDL buffer where duplication should start. It must be a multiple of the memory page size. • [in] `length` – size (in bytes) of the part of the original MDL buffer that needs to be duplicated. It must be a multiple of the memory page size. The following condition must also be fulfilled: *length<=size of original MDL buffer-offset*. • [out] `clone` – pointer to the handle of the created MDL buffer. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Preparing ELF image segments to be loaded into process memory

Information about API functions is provided in the table below.

Using the API

MDL buffers are used not only to create shared memory, but also to load ELF image segments into the memory of a new process. (ELF image segments are loaded by the `Einit` initializing program, for example.)

The `KnVmSegInitFromVm()` and `KnVmSegInitFromBuf()` functions create an MDL buffer and put the ELF image segment into it so that this segment can then be loaded into the memory of a new process.

Deleting MDL buffers containing ELF image segments

To delete MDL buffers containing ELF image segments, you must terminate the new process whose memory is mapped to these MDL buffers. You also need to complete the following steps before or after termination of this process:

1. Free the virtual memory regions that are mapped to the MDL buffers in the process that created these MDL buffers.

   This step is required only for MDL buffers that were created using the `KnVmSegInitFromVm()` function.

   To complete this step, use the `KnVmUnmap()` function.

2. Close the handles of MDL buffers in the process that created these MDL buffers.

   To complete this step, use the `KnHandleClose()` function, which is declared in the header file `sysroot-*-kos/include/coresrv/handle/handle_api.h` from the KasperskyOS SDK.

vmm_api.h functions

Function	Information about the function
`KnVmSegInitFromVm()`	Purpose Creates an MDL buffer from physical memory that is mapped to the defined virtual memory region containing the ELF image segment. Parameters • [out] `seg` – pointer to the structure describing the ELF image segment. This structure contains the handle of the MDL buffer (the `h` field) and is used to load an ELF image segment into process memory. The type of structure is defined in the header file `sysroot-*-kos/include/coresrv/vmm/vmm_types.h` from the KasperskyOS SDK. • [in,optional] `loadAddr` – page-aligned address for loading the ELF image segment into process memory, or `0` to select this address automatically. • [in] `addr` – page-aligned base address of the virtual memory region. • [in] `size` – size of the virtual memory region in bytes. It must be a multiple of the memory page size. • [in] `flags` – flags defining the access rights to the MDL buffer. The parameter type and flags are defined in the header file `sysroot-*-kos/include/vmm/flags.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information This function can be used if the defined virtual memory region is allocated with commitment. In the `flags` parameter, you can specify the following combinations of flags: • `VMM_FLAG_READ` – read access. • `VMM_FLAG_READ | VMM_FLAG_WRITE` – read-and-write access. • `VMM_FLAG_READ | VMM_FLAG_EXECUTE` – read-and-execute access. • `VMM_FLAG_RWX_MASK` or `VMM_FLAG_READ | VMM_FLAG_WRITE | VMM_FLAG_EXECUTE` – read/write/execute access.
`KnVmSegInitFromBuf()`	Purpose Creates an MDL buffer and copies the ELF image segment from the defined buffer to the MDL buffer. Parameters • [out] `seg` – pointer to the structure describing the ELF image segment. This structure contains the handle of the MDL buffer (the `h` field) and is used to load an ELF image segment into process memory. The type of structure is defined in the header file `sysroot-*-kos/include/coresrv/vmm/vmm_types.h` from the KasperskyOS SDK. • [in,optional] `loadAddr` – page-aligned address for loading the ELF image segment into process memory, or `0` to select this address automatically. • [in] `offset` – offset (in bytes) in the MDL buffer where writing of the ELF image segment must start. • [in] `size` – size (in bytes) of the ELF image segment. It must be a multiple of the memory page size. • [in] `flags` – flags defining the access rights to the MDL buffer. The parameter type and flags are defined in the header file `sysroot-*-kos/include/vmm/flags.h` from the KasperskyOS SDK. • [in] `fileOffset` – offset (in bytes) of the segment in the ELF image. • [in] `buildId` – pointer to the build ID. The linker writes this ID to the ELF image.

	• [in] `buildIdSize` – size (in bytes) of the build ID. • [in] `buf` – pointer to the buffer containing the ELF image segment. • [in] `bufSize` – size (in bytes) of the buffer containing the ELF image segment. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information In the `flags` parameter, you can specify the following combinations of flags: • `VMM_FLAG_READ` – read access. • `VMM_FLAG_READ | VMM_FLAG_WRITE` – read-and-write access. • `VMM_FLAG_READ | VMM_FLAG_EXECUTE` – read-and-execute access. • `VMM_FLAG_RWX_MASK` or `VMM_FLAG_READ | VMM_FLAG_WRITE | VMM_FLAG_EXECUTE` – read/write/execute access.

Allocating and freeing memory (alloc.h)

The API is defined in the header file `sysroot-*-kos/include/kos/alloc.h` from the KasperskyOS SDK.

The API is intended for allocating and freeing memory. Allocated memory is a committed virtual memory region that can be accessed for read-and-write operations.

Information about API functions is provided in the table below.

alloc.h functions

Function	Information about the function
`KosMemAllocEx()`	Purpose Allocates memory. Parameters • [in] `size` – size (in bytes) of the allocated memory. If you specify a size of zero, the function returns a pointer that can be passed to the `KosMemFree()` function. • [in] `align` – value defining the alignment of the allocated memory. It must be a power of two. The address of allocated memory can be unaligned (*align=1*) or aligned (*align=2,4,...,2^N*) to the boundary of a *2^N*-byte sequence (for example, two-byte or four-byte). • [in] `zeroed` – value defining the initialization of the allocated memory (`1` – initialize with zeros, `0` – do not initialize). Returned values If successful, the function returns the pointer to the allocated memory, otherwise it returns `RTL_NULL`.
`KosMemAlloc()`	Purpose Allocates memory. Parameters • [in] `size` – size (in bytes) of the allocated memory. If you specify a size of zero, the function returns a pointer that can be passed to the `KosMemFree()` function. Returned values

	If successful, the function returns the pointer to the allocated memory, otherwise it returns `RTL_NULL`.
`KosMemZalloc()`	Purpose Allocates memory and initializes it with zeros. Parameters • [in] `size` – size (in bytes) of the allocated memory. If you specify a size of zero, the function returns a pointer that can be passed to the `KosMemFree()` function. Returned values If successful, the function returns the pointer to the allocated memory, otherwise it returns `RTL_NULL`.
`KosMemFree()`	Purpose Deallocates memory. Parameters • [in] `ptr` – pointer to the freed memory. If you specify `RTL_NULL`, the function does not perform any actions. Returned values N/A
`KosMemRealloc()`	Purpose Allocates memory and (optionally) copies the contents of previously allocated memory to the newly allocated memory and frees the previously allocated memory after the copy operation is complete. Parameters • [in,optional] `ptr` – pointer to the previously allocated memory, or `RTL_NULL` if you only need to allocate memory without copying the contents of previously allocated memory. • [in] `size` – size (in bytes) of the newly allocated memory. The contents of previously allocated memory are copied to the newly allocated memory in whatever amount that can be accommodated in the newly allocated memory. If you specify a size of zero, the function returns a pointer that can be passed to the `KosMemFree()` function. Returned values If successful, the function returns the pointer to the allocated memory, otherwise it returns `RTL_NULL`.
`KosMemReallocEx()`	Purpose Allocates memory and (optionally) copies the contents of previously allocated memory to the newly allocated memory and frees the previously allocated memory after the copy operation is complete. Parameters • [in,optional] `ptr` – pointer to the previously allocated memory, or `RTL_NULL` if you only need to allocate memory without copying the contents of previously allocated memory. • [in] `size` – size (in bytes) of the newly allocated memory. The contents of previously allocated memory are copied to the newly allocated memory in whatever amount that can be accommodated in the newly allocated memory. If you specify a size of zero, the function returns a pointer that can be passed to the `KosMemFree()` function. • [in] `zeroed` – value defining the initialization of the allocated memory (`1` – initialize with zeros, `0` – do not initialize). If the `ptr` parameter is used to pass `RTL_NULL` and the `zeroed` parameter is set to `1`, the allocated memory will be initialized with zeros. If the size of the newly allocated memory is less than or equal to the size of the previously allocated memory, the value of the `zeroed` parameter does not affect the contents of the newly allocated memory. If the size of the newly allocated memory is greater than the size of the previously allocated memory, and the `zeroed` parameter is set to `1`, the fragment of newly allocated memory that is not occupied by the contents copied from the previously allocated memory may be initialized with zeros. To ensure that this fragment is initialized with zeros, you must do the following. The memory must be initially allocated by using either the `KosMemZalloc()` function, the `KosMemAllocEx()` function with the `zeroed` parameter set to `1`, or the `KosMemReallocEx()` function with the `zeroed` parameter set to `1` and the `ptr` parameter set to `RTL_NULL`. All subsequent `KosMemReallocEx()` calls must have the `zeroed` parameter set to `1`. Returned values

	If successful, the function returns the pointer to the allocated memory, otherwise it returns `RTL_NULL`.

Using DMA (dma.h)

The API is defined in the header file `sysroot-*-kos/include/coresrv/io/dma.h` from the KasperskyOS SDK.

The API is designed to set up data exchange between devices and RAM in *direct memory access* (DMA) mode in which the processor is not used.

Information about API functions is provided in the table below.

Using the API

The standard scenario for API usage includes the following steps:

1. Creating a DMA buffer.

   DMA buffer consists of one or more physical memory regions (blocks) that are used for DMA. A DMA buffer consisting of multiple blocks can be used if the device supports "scatter/gather DMA" mode. A DMA buffer consisting of one block can be used only if the device supports "scatter/gather DMA" or "continuous DMA" mode. The likelihood of creating a DMA buffer consisting of one large block is lower than the likelihood of creating a DMA buffer consisting of multiple small blocks. This is especially relevant when physical memory is highly fragmented.

   > If the device supports only "continuous DMA" mode, you must use a DMA buffer consisting of one block even if IOMMU is enabled.

   To complete this step, call the `KnIoDmaCreate()` or `KnIoDmaCreateContinuous()` function. The `KnIoDmaCreateContinuous()` function creates a DMA buffer consisting of one block. The `KnIoDmaCreate()` function creates a DMA buffer consisting of one block if the *2^order* value is equal to the *memory page size* value, or if the *2^order* value is the next largest value of the *memory page size* in the ascending ordered set *{2^(order-1);memory page size;2^order}*. If the value of the *memory page size* is greater than the *2^order* value, the `KnIoDmaCreate()` function can create a DMA buffer consisting of multiple blocks.

   The DMA buffer handle can be transferred to another process via IPC.

2. Mapping the DMA buffer to the memory of processes.

   One DMA buffer can be mapped to multiple virtual memory regions of one or more processes that own the handle of this DMA buffer. Mapping allows processes to receive read-and/or-write access to the DMA buffer.

   To reserve a virtual memory region and map the DMA buffer to it, call the `KnIoDmaMap()` function.

   A handle received when calling the `KnIoDmaMap()` function cannot be transferred to another process via IPC.

3. Opening access to the DMA buffer for a device via the `KnIoDmaBegin()` or `KnIoDmaBeginEx()` function call.

   The `KnIoDmaBegin()` or `KnIoDmaBeginEx()` function must be called to create a kernel object containing the addresses and sizes of blocks comprising the DMA buffer. A device needs this information to use the DMA buffer. A device can work with physical addresses and/or virtual addresses depending on whether IOMMU is enabled. If IOMMU is enabled, you must use the `KnIoDmaBegin()` or `KnIoDmaBeginEx()` function, and the object will contain virtual and physical addresses of blocks. If IOMMU is not enabled, you must use the `KnIoDmaBegin()` function, and the object will contain physical addresses of blocks only.

   When IOMMU is enabled, the device must be attached to an IOMMU domain.

   A handle received when calling the `KnIoDmaBegin()` or `KnIoDmaBeginEx()` function cannot be transferred to another process via IPC.

4. Information about the DMA buffer is received.

   At this step, get the addresses and sizes of blocks from the kernel object that was created by calling the `KnIoDmaBegin()` or `KnIoDmaBeginEx()` function. The received addresses and sizes will need to be passed to the device by using MMIO, for example. After receiving this information, the device can write to the DMA buffer and/or read from it.

   To complete this step, call one of the following functions:

   - `KnIoDmaGetInfo()`
   - `KnIoDmaGetPhysInfo()`
   - `KnIoDmaContinuousGetDmaAddr()`
   - `KnIoDmaContinuousGetPhysAddr()`

   The `KnIoDmaGetInfo()` and `KnIoDmaGetPhysInfo()` functions get the memory page number (`frame`) and the `order` for each block. (The memory page number multiplied by the memory page size results in the block address. The *2^order* value is the block size in memory pages.) These functions use the output parameter `outInfo` to pass the `KosObject` that contains the structure with information about the DMA buffer (for details about `KosObjects`, see Using KosObjects (objects.h)). The type of structure is defined in the header file `sysroot-*-kos/include/io/io_dma.h` from the KasperskyOS SDK. The `KnIoDmaGetInfo()` function inserts data into all fields of the structure, while the `KnIoDmaGetPhysInfo()` function inserts data only into the `count` and `descriptors` fields while writing zeros to all other fields. When using the `KnIoDmaGetInfo()` function, the `descriptors` array contains virtual memory page numbers if IOMMU is enabled, and it contains physical memory page numbers if IOMMU is not enabled. When using the `KnIoDmaGetPhysInfo()` function, the `descriptors` array contains physical memory page numbers regardless of whether or not IOMMU is enabled.

   The `KnIoDmaContinuousGetDmaAddr()` and `KnIoDmaContinuousGetPhysAddr()` functions can be used if the DMA buffer consists of one block. These functions let you get the block address. (The accepted block size should be the DMA buffer size that was defined when it was created.) The `KnIoDmaContinuousGetDmaAddr()` function uses the output parameter `addr` to pass the virtual address if IOMMU is enabled or the physical address if IOMMU is not enabled. The `KnIoDmaContinuousGetPhysAddr()` function uses the output parameter `addr` to pass the physical address, regardless of whether or not IOMMU is enabled.

   > On Raspberry Pi 4 B, the `KnIoDmaGetPhysInfo()` function lets you get the actual physical page numbers, while the `KnIoDmaGetInfo()` function lets you get the offset physical page numbers because some devices use Visual Processing Unit (VPU) translation for DMA. Similarly, the `KnIoDmaContinuousGetPhysAddr()` and `KnIoDmaContinuousGetDmaAddr()` functions let you get the actual and offset physical address, respectively.

Closing access to the DMA buffer for a device

Access to the DMA buffer can be closed only if IOMMU is enabled. If you delete the kernel object that was created when the `KnIoDmaBegin()` or `KnIoDmaBeginEx()` function was called, device access to the DMA buffer will be closed. To delete this object, you must close its handle.

Deleting a DMA buffer

To delete a DMA buffer, complete the following steps:

1. Free the virtual memory regions that were reserved during `KnIoDmaMap()` function calls.

To complete this step, you must close the handles that were received from `KnIoDmaMap()` function calls.

This step must be completed for all processes whose memory is mapped to the DMA buffer.

2. Delete the kernel object that was created by the `KnIoDmaBegin()` or `KnIoDmaBeginEx()` function call.

   To complete this step, you must close the handle that was received when the `KnIoDmaBegin()` or `KnIoDmaBeginEx()` function was called.

3. Close or revoke each DMA buffer handle in all processes that own these handles.

Information about API functions

dma.h functions

Function	Information about the function
`KnIoDmaCreate()`	Purpose Creates a DMA buffer. Parameters • [in] `order` – parameter defining the minimum number of memory pages (*2^order*) in a block. • [in] `size` – size (in bytes) of the DMA buffer. It must be a multiple of the memory page size. • [in] `flags` – flags defining the DMA buffer parameters. The parameter type and flags are defined in the header file `sysroot-*-kos/include/io/io_dma.h` from the KasperskyOS SDK. • [out] `outRid` – pointer to the DMA buffer handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information In the `flags` parameter, you can specify the following flags: • `DMA_DIR_TO_DEVICE` – the device has read-access to the DMA buffer. • `DMA_DIR_FROM_DEVICE` – the device has write-access to the DMA buffer. • `DMA_DIR_BIDIR` – the device has read-and-write access to the DMA buffer. • `DMA_ZONE_DMA32` – only the first four gigabytes of physical memory can be used to create a DMA buffer. • `DMA_ATTR_WRITE_BACK`, `DMA_ATTR_WRITE_THROUGH`, `DMA_ATTR_CACHE_DISABLE`, `DMA_ATTR_WRITE_COMBINE`, `DMA_RULE_CACHE_VOLATILE`, `DMA_RULE_CACHE_FIXED` – cache management.
`KnIoDmaCreateContinuous()`	Purpose Creates a DMA buffer consisting of one block. Parameters • [in] `size` – size (in bytes) of the DMA buffer. It must be a multiple of the memory page size. • [in] `flags` – flags defining the DMA buffer parameters. The parameter type and flags are defined in the header file `sysroot-*-kos/include/io/io_dma.h` from the KasperskyOS SDK. • [out] `outRid` – pointer to the DMA buffer handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

	Additional information In the `flags` parameter, you can specify the following flags: • `DMA_DIR_TO_DEVICE` – the device has read-access to the DMA buffer. • `DMA_DIR_FROM_DEVICE` – the device has write-access to the DMA buffer. • `DMA_DIR_BIDIR` – the device has read-and-write access to the DMA buffer. • `DMA_ZONE_DMA32` – only the first four gigabytes of physical memory can be used to create a DMA buffer. • `DMA_ATTR_WRITE_BACK`, `DMA_ATTR_WRITE_THROUGH`, `DMA_ATTR_CACHE_DISABLE`, `DMA_ATTR_WRITE_COMBINE`, `DMA_RULE_CACHE_VOLATILE`, `DMA_RULE_CACHE_FIXED` – cache management.
`KnIoDmaMap()`	Purpose Reserves a virtual memory region and maps the DMA buffer to it. Parameters • [in] `rid` – DMA buffer handle. • [in] `offset` – offset (in bytes) in the DMA buffer where mapping should start. It must be a multiple of the memory page size. • [in] `length` – size (in bytes) of the part of the DMA buffer that needs to be mapped. It must be a multiple of the memory page size. The following condition must also be fulfilled: *length<=size of DMA buffer-offset*. • [in,optional] `hint` – page-aligned, preferred base address of the virtual memory region, or `0` to select this address automatically. • [in] `vmflags` – flags defining the access rights to the virtual memory region. The flags are defined in the header file `sysroot-*-kos/include/vmm/flags.h` from the KasperskyOS SDK. • [out] `addr` – pointer to the base address of the virtual memory region. • [out] `handle` – pointer to the handle that is used to free the virtual memory region. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information In the `vmflags` parameter, you can specify the following flags: • `VMM_FLAG_READ` – read access. • `VMM_FLAG_WRITE` – write access.
`KnIoDmaModify()`	Purpose Modifies the DMA buffer cache settings. Parameters • [in] `rid` – DMA buffer handle. • [in] `newAttr` – flags defining the DMA buffer caching parameters. The flags are defined in the header file `sysroot-*-kos/include/io/io_dma.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information

	This function can be used if the following conditions are fulfilled: 1. The `DMA_RULE_CACHE_VOLATILE` flag was specified when the DMA buffer was created. 2. The DMA buffer is not mapped to virtual memory. 3. The `DMA_RULE_CACHE_VOLATILE` flag was specified during the previous function call (if completed). In the `newAttr` parameter, you can specify the following flags: • `DMA_ATTR_WRITE_BACK`, `DMA_ATTR_WRITE_THROUGH`, `DMA_ATTR_CACHE_DISABLE`, `DMA_ATTR_WRITE_COMBINE`, `DMA_RULE_CACHE_VOLATILE` – cache management.
`KnIoDmaGetInfo()`	Purpose Gets information about a DMA buffer. This information includes the addresses and sizes of blocks. Parameters • [in] `rid` – DMA buffer handle. • [out] `outInfo` – pointer to the address of the object containing the structure with information about the DMA buffer. The type of structure is defined in the header file `sysroot-*-kos/include/io/io_dma.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information To delete an object that was received via the `outInfo` parameter, you must use the `KosPutObject()` function from the objects.h API.
`KnIoDmaGetPhysInfo()`	Purpose Gets the physical addresses and sizes of DMA buffer blocks. Parameters • [in] `rid` – DMA buffer handle. • [out] `outInfo` – pointer to the address of the object containing the structure with information about the DMA buffer. The type of structure is defined in the header file `sysroot-*-kos/include/io/io_dma.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information To delete an object that was received via the `outInfo` parameter, you must use the `KosPutObject()` function from the objects.h API.
`KnIoDmaContinuousGetDmaAddr()`	Purpose Gets the block address for a DMA buffer consisting of one block. Parameters • [in] `rid` – DMA buffer handle. • [out] `addr` – pointer to the block address. The type of address is defined in the header file `sysroot-*-kos/include/hal/mtypes.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

`KnIoDmaContinuousGetPhysAddr()`	Purpose Gets the block physical address for a DMA buffer consisting of one block. Parameters • [in] `rid` – DMA buffer handle. • [out] `addr` – pointer to the physical block address. The type of address is defined in the header file `sysroot-*-kos/include/hal/mtypes.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnIoDmaBegin()`	Purpose Opens access to a DMA buffer for a device. Parameters • [in] `rid` – DMA buffer handle. • [out] `handle` – pointer to the handle of the kernel object containing the addresses and sizes of blocks that were required for the device to use the DMA buffer. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnIoDmaBeginEx()`	Purpose Opens access to a DMA buffer for a device. Parameters • [in] `rid` – DMA buffer handle. • [in] `domain` – IOMMU domain handle. • [out] `handle` – pointer to the handle of the kernel object containing the addresses and sizes of blocks that were required for the device to use the DMA buffer. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Memory-mapped I/O (mmio.h)

The API is defined in the header file `sysroot-*-kos/include/coresrv/io/mmio.h` from the KasperskyOS SDK.

The API is intended for working with MMIO memory. *MMIO memory* consists of physical addresses that are mapped to registers and memory of devices. (Portions of one physical address space are used for physical memory addressing and access to registers and memory of devices.)

Information about API functions is provided in the table below.

Using the API

The standard scenario for API usage includes the following steps:

1. The MMIO memory region corresponding to the relevant device is registered.

   To complete this step, call the `KnRegisterPhyMem()` function.

The handle of an MMIO memory region can be transferred to another process via IPC.

2. Mapping the MMIO memory region to the memory of a process.

   An MMIO memory region can only be mapped to one virtual memory region of only one process at one time. (However, an MMIO memory region may be mapped to the virtual memory of different processes that own the handle of this region at different times.) Mapping allows the process to receive read-and/or-write access to the MMIO memory region.

   To reserve a virtual memory region and map the MMIO memory region to it, call the `KnIoMapMem()` function.

   A handle received when calling the `KnIoMapMem()` function cannot be transferred to another process via IPC.

3. Reading data from the MMIO memory region and writing data to it via process memory.

   The 8-, 16- and 32-bit words that are read from the MMIO memory region or written to it are values of device registers or contents of device memory.

   To complete this step, use the `IoReadMmReg8|16|32()`, `IoReadMmBuffer8|16|32()`, `IoWriteMmReg8|16|32()`, and `IoWriteMmBuffer8|16|32()` functions.

Deregistering an MMIO memory region

To deregister an MMIO memory region, complete the following steps:

1. Free the virtual memory region that was allocated when the `KnIoMapMem()` function was called.

   To complete this step, call the `KnHandleClose()` function and specify the handle that was received when the `KnIoMapMem()` function was called. (The `KnHandleClose()` function is declared in the header file `sysroot-*-kos/include/coresrv/handle/handle_api.h` from the KasperskyOS SDK.)

2. Close or revoke each MMIO memory region handle in all processes that own these handles.

   To complete this step, use the `KnHandleClose()` and/or `KnHandleRevoke()` functions that are declared in the header file `sysroot-*-kos/include/coresrv/handle/handle_api.h` from the KasperskyOS SDK.

Information about API functions

mmio.h functions

Function(s)	Information about the function(s)
`KnRegisterPhyMem()`	Purpose Registers an MMIO memory region. Parameters • [in] `addr` – page-aligned base address of the MMIO memory region. • [in] `size` – size of the MMIO memory region. It must be a multiple of the memory page size. • [out] `outRid` – pointer to the handle of the MMIO memory region. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnIoMapMem()`	Purpose Reserves the virtual memory region and maps the MMIO memory region to it. Parameters • [in] `rid` – handle of the MMIO memory region.

	• [in] `prot` – flags defining the access rights to the virtual memory region. The flags are defined in the header file `sysroot-*-kos/include/vmm/flags.h` from the KasperskyOS SDK. • [in] `attr` – flags defining the cache settings. The flags are defined in the header file `sysroot-*-kos/include/vmm/flags.h` from the KasperskyOS SDK. • [out] `addr` – pointer to the base address of the virtual memory region. • [out] `handle` – pointer to the handle that is used to free the virtual memory region. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information In the `prot` parameter, you can specify the following flags: • `VMM_FLAG_READ` – read access. • `VMM_FLAG_WRITE` – write access. In the `attr` parameter, you can specify the following flags: • `VMM_FLAG_WRITE_BACK`, `VMM_FLAG_WRITE_THROUGH`, `VMM_FLAG_WRITE_COMBINE`, `VMM_FLAG_CACHE_DISABLE` – cache management. It is recommended to specify the `VMM_FLAG_CACHE_DISABLE` flag.

`IoReadMmReg8()`
`IoReadMmReg16()`
`IoReadMmReg32()`
	Purpose
	
	Get an 8-, 16- or 32-bit word from an MMIO memory region.
	
	Parameters
	• [in] `reg` – virtual address for reading the word.
	
	Returned values
	
	8-, 16- or 32-bit word from the MMIO memory region.

`IoReadMmBuffer8()`
`IoReadMmBuffer16()`
`IoReadMmBuffer32()`
	Purpose
	
	Save a sequence of 8-, 16- or 32-bit words from the MMIO memory region to the buffer.
	
	Parameters
	• [in] `baseReg` – virtual address of the first word in the sequence.
	• [out] `dst` – pointer to the buffer used to store the sequence of words.
	• [in] `cnt` – number of words in the sequence.
	
	Returned values
	
	N/A
	
	Additional information
	
	The functions can be used if the processor has an x86 or x86-64 architecture.

`IoWriteMmReg8()`
`IoWriteMmReg16()`
`IoWriteMmReg32()`
	Purpose
	
	Write an 8-, 16- or 32-bit word to an MMIO memory region.
	
	Parameters
	• [in] `reg` – virtual address for writing the word.
	• [in] `data` – word to write.
	
	Returned values

	N/A

`IoWriteMmBuffer8()`
`IoWriteMmBuffer16()`
`IoWriteMmBuffer32()`
	Purpose
	
	Write a sequence of 8-, 16- or 32-bit words from the buffer to an MMIO memory region.
	
	Parameters
	
	• [in] `baseReg` – virtual address for writing the sequence of words.
	• [in] `src` – pointer to the buffer containing the sequence of words.
	• [in] `cnt` – number of words in the sequence.
	
	Returned values
	
	N/A
	
	Additional information
	
	The functions can be used if the processor has an x86 or x86-64 architecture.

Managing interrupt processing (irq.h)

The API is defined in the header file `sysroot-*-kos/include/coresrv/io/irq.h` from the KasperskyOS SDK.

The API manages the handling of hardware interrupts. A *hardware interrupt* is a signal sent from a device to direct the processor to immediately pause execution of the current program and instead handle an event related to this device. For example, pressing a key on the keyboard invokes a hardware interrupt that ensures the required response to this pressed key (for example, input of a character).

A hardware interrupt occurs when the device queries the interrupt controller. This query can be transmitted through a hardware interrupt line between the device and the interrupt controller or through MMIO memory. In the second case, the device writes to MMIO memory by calling the *Message Signaled Interrupt* (MSI).

> At present, no functions for managing the handling of MSI interrupts have been implemented.

Each hardware interrupt line corresponds to one interrupt with a unique number.

Information about API functions is provided in the table below.

Using the API

To attach an interrupt to its handler, complete the following steps:

1. Registering an interrupt by calling the `KnRegisterIrq()` function.
   One interrupt can be registered multiple times in one or more processes.
   The handle of an interrupt can be transferred to another process via IPC.
2. Attaching a thread to an interrupt by calling the `KnIoAttachIrq()` function.
   This step is performed by the thread in whose context the interrupt will be handled.

When using the handle received from the `KnRegisterIrq()` function call, you can attach only one thread to an interrupt. To attach multiple threads in one or more processes to an interrupt, use different handles for this interrupt received from separate `KnRegisterIrq()` function calls. In this case, the `KnIoAttachIrq()` function must be called with the same flags in the `flags` parameter.

A handle received when calling the `KnIoAttachIrq()` function cannot be transferred to another process via IPC.

To deny (mask) an interrupt, call the `KnIoDisableIrq()` function. To allow (unmask) an interrupt, call the `KnIoEnableIrq()` function. Even though these functions receive an interrupt handle that is used to attach only one thread to the interrupt, their action is applied to all threads that are attached to this interrupt. These functions must be called outside of the threads attached to the interrupt. After an interrupt is registered and a thread is attached to it, this interrupt does not require unmasking.

To initiate detachment of a thread from an interrupt, call the `KnIoDetachIrq()` function outside of the thread that is attached to the interrupt. Detachment is performed by the thread attached to the interrupt by calling the `KnThreadDetachIrq()` function declared in the header file `sysroot-*-kos/include/coresrv/thread/thread_api.h` from the KasperskyOS SDK.

Handling an interrupt

After attaching to an interrupt, a thread calls the `Call()` function declared in the header file `sysroot-*-kos/include/coresrv/syscalls.h` from the KasperskyOS SDK. The thread is locked as a result of this call. When an interrupt occurs or the `KnIoDetachIrq()` function is called, the KasperskyOS kernel sends an IPC message to the process that contains this thread. This IPC message contains a request to handle the interrupt or a request to detach the thread from the interrupt. When a process receives an IPC message, the `Call()` function in the thread attached to the interrupt returns control and provides the contents of the IPC message to the thread. The thread extracts the request from the IPC message and either processes the interrupt or detaches from the interrupt. If the interrupt is processed, information about its failure or success upon completion is added to the response IPC message that is sent to the kernel by the next `Call()` function call in the loop.

When processing an interrupt, use the `IoGetIrqRequest()` and `IoSetIrqAnswer()` functions that are declared in the header file `sysroot-*-kos/include/io/io_irq.h` from the KasperskyOS SDK. These functions let you extract data from IPC messages and add data to IPC messages for data exchange between the kernel and the thread attached to the interrupt.

The standard interrupt processing loop includes the following steps:

1. Adding information about the failure or success of interrupt processing to an IPC message by calling the `IoSetIrqAnswer()` function.

2. Sending the IPC message to the kernel and receiving an IPC message from the kernel.

   To complete this step, call the `Call()` functions. In the `handle` parameter, you must specify the handle that was received when the `KnIoAttachIrq()` function was called. You must use the `msgOut` parameter to define the IPC message that will be sent to the kernel, and use the `msgIn` parameter to define the IPC message that will be received from the kernel.

3. Extracting a request from the IPC message received from the kernel by calling the `IoGetIrqRequest()` function.

4. Processing the interrupt or detaching from the interrupt depending on the request.

   If the request requires detachment from the interrupt, exit the interrupt processing loop and call the `KnThreadDetachIrq()` function.

Deregistering an interrupt

To deregister an interrupt, complete the following steps:

1. Detach the thread from the interrupt.

   To complete this step, call the `KnThreadDetachIrq()` function.

2. Close the handle that was received when the `KnIoAttachIrq()` function was called.

   To complete this step, call the `KnHandleClose()` function. (The `KnHandleClose()` function is declared in the header file `sysroot-*-kos/include/coresrv/handle/handle_api.h` from the KasperskyOS SDK.)

3. Close or revoke each interrupt handle in all processes that own these handles.

   To complete this step, use the `KnHandleClose()` and/or `KnHandleRevoke()` functions that are declared in the header file `sysroot-*-kos/include/coresrv/handle/handle_api.h` from the KasperskyOS SDK.

One interrupt can be registered multiple times, but completion of these steps cancels only one registration. The other registrations will remain active. Each registration of one interrupt must be canceled separately.

Information about API functions

irq.h functions

Function	Information about the function
`KnRegisterIrq()`	Purpose Registers an interrupt. Parameters • [in] `irq` – interrupt number. • [out] `outRid` – pointer to the interrupt handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnIoAttachIrq()`	Purpose Attaches the calling thread to an interrupt. Parameters • [in] `rid` – interrupt handle. • [in] `flags` – flags defining the interrupt parameters. Flags are defined in the header files `sysroot-*-kos/include/io/io_irq.h` and `sysroot-*-kos/include/hal/irqmode.h` from the KasperskyOS SDK. • [out] `handle` – pointer to the client IPC handle that is used by the interrupt handler. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information In the `flags` parameter, you can specify the following flags: • `IRQ_LEVEL_LOW` – the interrupt occurs when the signal level is low. • `IRQ_LEVEL_HIGH` – the interrupt occurs when the signal level is high. • `IRQ_EDGE_RAISE` – the interrupt occurs when the signal level increases. • `IRQ_EDGE_FALL` – the interrupt occurs when the signal level decreases.

	• `IRQ_PRIO_LOW` – the interrupt has low priority. • `IRQ_PRIO_NORMAL` – the interrupt has medium priority. • `IRQ_PRIO_HIGH` – the interrupt has high priority. • `IRQ_PRIO_RT` – the interrupt has the highest priority.
`KnIoDetachIrq()`	Purpose Sends a request to a thread. When this request is fulfilled, the thread must detach from the interrupt. Parameters • [in] `rid` – interrupt handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnIoEnableIrq()`	Purpose Allows (unmasks) an interrupt. Parameters • [in] `rid` – interrupt handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnIoDisableIrq()`	Purpose Denies (masks) an interrupt. Parameters • [in] `rid` – interrupt handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Managing threads (high-level API thread.h)

The API is defined in the header file `sysroot-*-kos/include/kos/thread.h` from the KasperskyOS SDK.

Main capabilities of the API:

- Create, terminate, and lock threads.
- Resume execution of locked threads.
- Register functions that are called when creating and terminating threads.

Information about API functions is provided in the table below.

> The `libkos` library also provides a low-level API for thread management. This API is defined in the header file `sysroot-*-kos/include/coresrv/thread/thread_api.h` from the KasperskyOS SDK. The low-level API should be used only if the high-level API does not have sufficient capabilities.

Creating threads

To create a thread, call the `KosThreadCreate()` or `KosThreadCreateDetached()` function. These functions create a standard thread with priority ranging from 0–15. (For details about standard threads, see Managing threads (low-level API thread_api.h)).

Terminating threads

When a thread is terminated, it is permanently deleted from the scheduler. A thread is terminated in the following cases:

- The function executed by the thread is exited.

  The root function (not a nested function) performed by the thread must be exited by the `return` operator.

- The `KosThreadTerminate()` or `KosThreadExit()` function is called.

  The `KosThreadExit()` function terminates the thread even if the function is called from a nested function instead of the root function executed by the thread.

- The process was terminated or switched to the "frozen" state.

  For details about the "frozen" state, see Managing processes (low-level API task_api.h).

Thread exit codes are defined by the developer of the KasperskyOS-based solution. These codes should be specified in the `exitCode` parameter of the `KosThreadTerminate()` and `KosThreadExit()` functions, and when calling the `return` operator in the function executed by the thread. To get the thread exit code, call the `KosThreadWait()` function. If successful, this function returns the thread exit code. Otherwise, it returns `-1`. Therefore, thread exit codes must differ from `-1` to avoid ambiguity.

Registering functions that are called when creating and terminating threads

To register a function that is called when threads of a process are created and terminated, call the `KosThreadCallbackRegister()` function from this process.

The registered function is called in the following cases:

- A thread is created by calling the `KosThreadCreate()` function.

- The thread that was created by calling the `KosThreadCreate()` function is terminated as a result of exiting a function executed by this thread.

- The thread is terminated by calling the `KosThreadExit()` function.

You can register multiple functions, and each of them will be called when a thread is created and terminated. When a thread is created, the registered function is called with the `KosThreadCallbackReasonCreate` argument in the `reason` parameter. When the thread is terminated, the registered function is called with the `KosThreadCallbackReasonDestroy` argument in the `reason` parameter.

Guaranteeing that a function can only be called once

The callback function defined through the `initRoutine` parameter is called only when the `KosThreadOnce()` function is called for the first time. This does not occur on repeated calls of the `KosThreadOnce()` function (even from other threads), and the `KosThreadOnce()` function simply returns control. For example, this ensures that a driver is initialized only once when multiple software components use this driver and start its initialization irrespective of each other.

Special considerations for threads attached to interrupts

After a thread is attached to an interrupt, the thread becomes a real-time thread with a FIFO scheduler class and a priority higher than 31. (For details about attaching a thread to an interrupt, see Managing interrupt processing (irq.h). For details about the FIFO real-time thread scheduler class, see Managing threads (low-level API thread_api.h)).

> Many API functions cannot be applied for threads that are attached to interrupts. These functions include `KosThreadCreate()`, `KosThreadSuspend()`, `KosThreadResume()`, `KosThreadTerminate()`, `KosThreadWait()`, `KosThreadSleep()`, and `KosThreadYield()`.

Defining and getting the base address of the TLS for threads

A *thread local storage* (TLS) is process memory in which a thread can store data in isolation from other threads. The base address for TLS is defined and received by the `KosThreadTlsSet()` and `KosThreadTlsGet()` functions, respectively. These functions are intended for use by the `libc` library.

Getting the base address and stack limit of threads

The `KosThreadGetStack()` function gets the base address and stack limit of the thread. This function is intended for use by the `libc` library.

Freeing resources of terminated threads

Resources of a thread include its stack, context, and TCB (for details about a TCB, see Managing threads (low-level API thread_api.h)). To release thread resources after the thread terminates, call the `KosThreadWait()` function to wait for completion of the thread. (Exceptions to this case are the initial thread of a process and a thread created by the `KosThreadCreateDetached()` function.) Thread resources are also released upon termination of the process that includes these threads.

Information about API functions

thread.h functions

Function	Information about the function
`KosThreadCallbackRegister()`	Purpose Registers the function that is called when creating and terminating threads of the calling process. Parameters • [in] `callback` – pointer to the function. Returned values

	If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosThreadCallbackUnregister()`	Purpose Deregisters the function that is called when creating and terminating threads of the calling process. Parameters • [in] `callback` – pointer to the function. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosThreadCreate()`	Purpose Creates a thread. Parameters • [out] `handle` – pointer to the thread handle. • [in] `priority` – thread priority. • [in,optional] `stackSize` – thread stack limit (in bytes), or `0` to use the default size that was defined when the process was created. • [in] `routine` – pointer to the function executed by the thread. The parameter type is defined in the header file `sysroot-*-kos/include/coresrv/thread/thread_api.h` from the KasperskyOS SDK. • [in,optional] `context` – pointer to the parameters passed to the function defined through the `routine` parameter, or `RTL_NULL` if parameters do not need to be passed to this function. • [in] `suspended` – value defining whether the thread will be created in a locked state (`1`) or unlocked state (`0`). Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosThreadCreateDetached()`	Purpose Creates an unlocked thread and closes its handle. Parameters • [in] `priority` – thread priority. • [in,optional] `stackSize` – thread stack limit (in bytes), or `0` to use the default size that was defined when the process was created. • [in] `routine` – pointer to the function executed by the thread. The parameter type is defined in the header file `sysroot-*-kos/include/coresrv/thread/thread_api.h` from the KasperskyOS SDK. • [in,optional] `context` – pointer to the parameters passed to the function defined through the `routine` parameter, or `RTL_NULL` if parameters do not need to be passed to this function. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosThreadCurrentId()`	Purpose Gets the ID (TID) of the calling thread. Parameters N/A Returned values

	Thread ID. The ID type is defined in the header file `sysroot-*-kos/include/thread/tidtype.h` from the KasperskyOS SDK.
`KosThreadSuspend()`	Purpose Locks the calling thread. Parameters • [in] `handle` – thread handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosThreadResume()`	Purpose Resumes execution of a locked thread. Parameters • [in] `handle` – thread handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosThreadExit()`	Purpose Terminates the calling thread. Parameters • [in] `exitCode` – thread exit code. Returned values N/A
`KosThreadWait()`	Purpose Locks the calling thread until the defined thread is terminated. Parameters • [in] `handle` – thread handle. • [in] `timeout` – timeout (in milliseconds) for the termination of a thread, or `INFINITE_TIMEOUT` to define an unlimited timeout. Returned values If successful, the function returns the thread exit code, otherwise it returns `-1`.
`KosThreadSleep()`	Purpose Locks the calling thread for the specified duration. Parameters • [in] `mdelay` – thread lockout duration (in milliseconds). Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosThreadYield()`	Purpose Gives up the time slice of the calling thread to the next thread in the queue. Parameters N/A Returned values

	N/A Additional information The `KosThreadYield()` function call is identical to the `KosThreadSleep()` function call with a zero value for the `mdelay` parameter.
`KosThreadTerminate()`	Purpose Terminates a thread. Parameters • [in] `handle` – thread handle. • [in] `exitCode` – thread exit code. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosThreadTlsGet()`	Purpose Gets the base address of the thread local storage (TLS) for the calling thread. Parameters N/A Returned values Pointer to the TLS, or `RTL_NULL` if the thread does not have a TLS.
`KosThreadTlsSet()`	Purpose Defines the base address of the thread local storage (TLS) for the calling thread. Parameters • [in] `tls` – pointer to the TLS. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosThreadGetStack()`	Purpose Gets the base address and stack limit of the thread. Parameters • [in] `handle` – thread handle. • [out] `size` – pointer to the stack limit (in bytes). Returned values Pointer to the thread stack.
`KosThreadOnce()`	Purpose Guarantees that the defined function will be called only once. Parameters • [in] `onceControl` – pointer to the variable that indicates whether or not the defined function was already called. This variable must be initialized with the `KOS_THREAD_ONCE_INIT` value. • [in] `initRoutine` – pointer to the function that must be called only one time. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Managing threads (low-level API thread_api.h)

The API is defined in the header file `sysroot-*-kos/include/coresrv/thread/thread_api.h` from the KasperskyOS SDK.

Main capabilities of the API:

- Create, terminate, and lock threads.
- Resume execution of locked threads.
- Change the priorities and scheduler classes of threads.
- Handle exceptions.
- Detach threads from interrupts.
- Attach threads to processors (processor cores).

> API functions that take the thread ID (TID) as input or return it are obsolete and will be removed in the future. These functions are not described in this section.

Information about API functions is provided in the table below.

> The `libkos` library also provides a high-level API for thread management. This API is defined in the header file `sysroot-*-kos/include/kos/thread.h` from the KasperskyOS SDK. It is recommended to use this specific API. The low-level API should be used only if the high-level API does not have sufficient capabilities.

Creating threads. Thread scheduler classes

To create a thread, call the `KnThreadCreateByHandle()` function. You can use the `flags` parameter of this function for the created thread to define the following scheduler classes:

- Class for scheduling standard threads.
- Class for scheduling FIFO real-time threads (First In, First Out).
- Class for scheduling RR real-time threads Round-Robin).

The priority of a standard thread can take values ranging from 0 to 15. The higher the priority of a standard thread, the larger the time slice allocated to this thread and the more frequently these time slices are allocated. One standard thread with a high priority can occupy multiple consecutive places in the thread queue. Standard threads cannot supersede other standard threads or real-time threads, regardless of their respective priorities. If a real-time thread appears in the queue, the current standard thread immediately transfers control to the real-time thread. If there are no real-time threads in the queue, the current standard thread transfers control to the next standard thread if the current one is terminated, becomes locked, uses up its time slice or gives it up to the next standard thread.

The priority of a real-time thread can take values ranging from 0 to 31. Higher-priority real-time threads supersede lower-priority real-time threads. Real-time threads also supersede standard threads, regardless of their respective priorities. Control is transferred from the current real-time thread to the next real-time thread in the queue in the following cases:

- The current real-time thread with a FIFO scheduler class is terminated or locked, or a real-time thread with a higher priority appears in the queue.

  Until the real-time thread with a FIFO scheduler class is terminated, locked, or superseded by a real-time thread with a higher priority, it can retain control indefinitely and not transfer control to the next real-time thread.

- The current real-time thread with an RR scheduler class is terminated, locked or uses up its time slice, or a real-time thread with a higher priority appears in the queue.

  A real-time thread with an RR scheduler class that is not terminated, locked, or superseded by a higher-priority real-time thread transfers control to the next one upon expiry of its time slice if the next one is a real-time thread with the same priority. Otherwise, it transfers control to itself again.

- The current real-time thread has given up its time slice to the next real-time thread with the same priority in the queue.

  A real-time thread can give up its time slice to the next one in the queue if the next one is a real-time thread with the same priority. Otherwise, it transfers control to itself again.

Execution of lower-priority real-time threads begins only after each of the real-time threads with a higher priority is terminated or locked. Real-time threads with the same priority form a queue based on the FIFO principle.

The thread scheduler class can be changed by calling the `KnThreadSetSchedPolicyByHandle()` function. This function also changes the size of the time slice allocated to a real-time thread with an RR scheduler class. This value is the only parameter of a real-time thread RR scheduler class that has a default value of 10 ms but can take values ranging from 2 ms to 100 ms. The scheduler class of standard threads and the scheduler class of FIFO real-time threads do not have parameters.

The priority of a thread can be changed by calling the `KnThreadSetSchedPolicyByHandle()` or `KnThreadSetPriorityByHandle()` function.

After a thread is attached to an interrupt, the thread becomes a real-time thread with a FIFO scheduler class and a priority higher than 31, irrespective of the specific scheduler class and priority this thread had before it was attached to the interrupt. Managing interrupt processing (irq.h).)

> Many API functions cannot be applied for threads that are attached to interrupts. These functions include `KnThreadCreateByHandle()`, `KnThreadSetPriorityByHandle()`, `KnThreadSuspendCurrent()`, `KnThreadResumeByHandle()`, `KnThreadTerminateByHandle()`, `KnThreadWaitByHandle()`, `KnSleep()`, and `KnThreadSetSchedPolicyByHandle()`.

Creating thread handles

A thread handle is created whenever a thread is created by the `KnThreadCreateByHandle()` function call. A thread (including the initial thread) can also create its own handle by calling the `KnThreadOpenCurrent()` function.

A thread handle cannot be transferred to another process via IPC.

Handling exceptions

To register an exception handling function for a thread, call the `KnThreadSetExceptionHandler()` function. This function deregisters the previous exception handling function and returns its ID. By saving this ID, you can subsequently register the previous exception handling function again.

To get information about the last exception of a thread, call the `KnThreadGetLastException()` function in the exception handler.

Detaching threads from interrupts

To detach a thread from an interrupt, call the `KnThreadDetachIrq()` function. (For more details about using the `KnThreadDetachIrq()` function, see Managing interrupt processing (irq.h).)

After a thread is detached from an interrupt, the thread receives the scheduler class and priority that it had before it was attached to the interrupt.

Attaching threads to processors (processor cores)

To restrict the set of processors (processor cores) that can be used to execute a thread, define an affinity mask for this thread. An *affinity mask* is a bit mask indicating which specific processors or processor cores must execute the thread.

To create, adjust, and perform other operations with affinity masks, use the API that is defined in the header file `sysroot-*-kos/include/rtl/cpuset.h` from the KasperskyOS SDK.

To define a thread affinity mask, call the `KnThreadSetAffinityByHandle()` function.

Terminating threads

When a thread is terminated, it is permanently deleted from the scheduler. A thread is terminated in the following cases:

- The function executed by the thread is exited.

  The root function (not a nested function) performed by the thread must be exited by the `return` operator.

- The `KnThreadTerminateByHandle()` or `KnThreadExit()` function is called.

  The `KnThreadExit()` function terminates the thread even if the function is called from a nested function instead of the root function executed by the thread.

- The process was terminated or switched to the "frozen" state.

  For details about the "frozen" state, see Managing processes (low-level API task_api.h).

Thread exit codes are defined by the developer of the KasperskyOS-based solution. These codes should be specified in the `code` parameter of the `KnThreadTerminateByHandle()` and `KnThreadExit()` functions, and when calling the `return` operator in the function executed by the thread. To get the thread exit code, call the `KnThreadWaitByHandle()` function.

Getting the address of the TCB and defining the base address of a TLS for threads

A *thread control block* (TCB) is a structure containing thread information that is used by the kernel to manage this specific thread. The type of structure is defined in the header file `sysroot-*-kos/include/thread/tcbpage.h` from the KasperskyOS SDK. A TCB contains the base address of the thread local storage (TLS). The `KnThreadGetTcb()` function gets the TCB address. The `KnThreadSetTls()` function defines the base address of the TLS. These functions are intended for use by the `libc` library.

Getting thread information

The `KnThreadGetInfoByHandle()` function gets the base address and stack limit of the thread, and the thread identifier (TID). This function is intended for use by the `libc` library.

Freeing resources of terminated threads

Resources of a thread include its stack, context, and TCB. To free the resources of a thread after the thread is terminated, you need to close its handle before or after termination of this thread by calling the `KnHandleClose()` function, which is declared in the header file `sysroot-*-kos/include/coresrv/handle/handle_api.h` from the KasperskyOS SDK (an exception to this case is the initial thread of a process whose resources are freed without calling the `KnHandleClose()` function if the initial thread did not create its own handle by calling the `KnThreadOpenCurrent()` function). Resources of threads are also freed upon termination of the process that includes these threads.

Information about API functions

thread_api.h functions

Function	Information about the function
`KnThreadCreateByHandle()`	Purpose Creates a thread. Parameters • [out,optional] `thread` – pointer to the thread handle. Specifying `RTL_NULL` closes the handle automatically after the thread is created. • [in] `priority` – thread priority. • [in,optional] `stackSize` – thread stack limit (in bytes), or `0` to use the default size that was defined when the process was created. • [in] `startRoutine` – parameter that must have the value `RTL_NULL`. • [in] `routine` – pointer to the function executed by the thread. • [in,optional] `context` – pointer to the parameters passed to the function defined through the `routine` parameter, or `RTL_NULL` if parameters do not need to be passed to this function. • [in] `flags` – flags defining the parameters for creating the thread. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information In the `flags` parameter, you can specify the following flags: • `ThreadFlagCreateSuspended` – creates a thread in the locked state. • `ThreadFlagCreateSchedOther` – creates a standard thread.

	• `ThreadFlagCreateSchedFifo` – creates a real-time thread with a FIFO scheduler class. • `ThreadFlagCreateSchedRR` – creates a real-time thread with an RR scheduler class. A standard thread is created by default. Handles of threads cannot be transferred between processes via IPC.
`KnThreadOpenCurrent()`	Purpose Creates the handle of the calling thread. Parameters • [out] `thread` – pointer to the thread handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnThreadGetPriorityByHandle()`	Purpose Gets the priority of a thread. Parameters • [in] `thread` – thread handle. • [out] `priority` – pointer to the thread priority. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnThreadSetPriorityByHandle()`	Purpose Defines the priority of a thread. Parameters • [in] `thread` – thread handle. • [in] `priority` – thread priority. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnThreadSuspendCurrent()`	Purpose Locks the calling thread. Parameters N/A Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnThreadResumeByHandle()`	Purpose Resumes execution of a locked thread. Parameters • [in] `thread` – thread handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnThreadTerminateByHandle()`	Purpose Terminates a thread.

	Parameters • [in] `thread` – thread handle. • [in] `code` – thread exit code. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnThreadExit()`	Purpose Terminates the calling thread. Parameters • [in] `code` – thread exit code. Returned values Error code.
`KnThreadGetInfoByHandle()`	Purpose Gets information about a thread. Parameters • [in] `thread` – thread handle. • [out] `info` – pointer to the structure containing the base address of the thread stack and its size (in bytes), and the thread identifier (TID). Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnThreadWaitByHandle()`	Purpose Locks the calling thread until the defined thread is terminated. Parameters • [in] `thread` – thread handle. • [in] `msDelay` – timeout (in milliseconds) for the termination of a thread, or `INFINITE_TIMEOUT` to define an unlimited timeout. • [out] `exitCode` – pointer to the thread exit code. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnSleep()`	Purpose Locks the calling thread for the specified duration. Parameters • [in] `mdelay` – thread lockout duration (in milliseconds). Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnThreadGetTcb()`	Purpose Gets the address of the thread control block (TCB) for the calling thread. Parameters N/A

	Returned values Pointer to the TCB. The data type for TCB storage is defined in the header file `sysroot-*-kos/include/thread/tcbpage.h` from the KasperskyOS SDK.
`KnThreadSetTls()`	Purpose Defines the base address of the thread local storage (TLS) for the calling thread. Parameters • [in] `tls` – pointer to the TLS. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnThreadDetachIrq()`	Purpose Detaches the calling thread from the interrupt handled in its context. Parameters N/A Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnThreadSetExceptionHandler()`	Purpose Registers the exception handling function for the calling thread. Parameters • [in,optional] `handler` – ID of the exception handling function, or `RTL_NULL` to deregister the previously registered exception handling function without registering a new one. The parameter type is defined in the header file `sysroot-*-kos/include/thread/tcbpage.h` from the KasperskyOS SDK. Returned values ID of the previously registered exception handling function if one exists, otherwise `RTL_NULL`. The type of returned value is defined in the header file `sysroot-*-kos/include/thread/tcbpage.h` from the KasperskyOS SDK.
`KnThreadGetLastException()`	Purpose Gets information about the last exception of the calling thread. Parameters • [out] `exception` – pointer to the structure containing information about the exception. The type of structure is defined in the header file `sysroot-*-kos/include/thread/tcbpage.h` from the KasperskyOS SDK. Returned values N/A
`KnThreadGetAffinityByHandle()`	Purpose Gets a thread affinity mask. Parameters • [in] `thread` – thread handle. • [out] `mask` – pointer to the structure containing the thread affinity mask. The type of structure is defined in the header file `sysroot-*-kos/include/rtl/cpuset.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

`KnThreadSetAffinityByHandle()`	Purpose Defines a thread affinity mask. Parameters • [in] `thread` – thread handle. • [in] `mask` – pointer to the structure containing the thread affinity mask. The type of structure is defined in the header file `sysroot-*-kos/include/rtl/cpuset.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnThreadGetSchedPolicyByHandle()`	Purpose Gets information about the thread scheduler class. Parameters • [in] `thread` – thread handle. • [out] `policy` – pointer to the thread scheduler class. The data type for storing the thread scheduler class is defined in the header file `sysroot-*-kos/include/thread/tidtype.h` from the KasperskyOS SDK. • [out,optional] `param` – pointer to the parameters of the thread scheduler class, or `RTL_NULL`. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information In the `param` parameter, you can specify `RTL_NULL` if you do not need to get the thread scheduler class parameters. In the `param` parameter, you must specify `RTL_NULL` if the thread scheduler class does not have any parameters.
`KnThreadSetSchedPolicyByHandle()`	Purpose Defines the scheduler class and priority of the thread. Parameters • [in] `thread` – thread handle. • [in] `policy` – thread scheduler class. The parameter type is defined in the header file `sysroot-*-kos/include/thread/tidtype.h` from the KasperskyOS SDK. • [in] `prio` – thread priority. • [in,optional] `param` – pointer to the parameters of the thread scheduler class, or `RTL_NULL`. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information In the `param` parameter, you can specify `RTL_NULL` if you need to define the thread scheduler class parameters that are applied by default. In the `param` parameter, you must specify `RTL_NULL` if the thread scheduler class does not have any parameters.

Managing processes (high-level API task.h)

The API is defined in the header file `sysroot-*-kos/include/kos/task.h` from the KasperskyOS SDK.

The API lets you create, start and terminate processes, and statically create IPC channels and pass handles.

Information about API functions is provided in the table below.

> The `libkos` library also provides a low-level API for process management. This API is defined in the header file `sysroot-*-kos/include/coresrv/task/task_api.h` from the KasperskyOS SDK. The low-level API should be used only if the high-level API does not have sufficient capabilities.

Creating processes

To create a process, call one of the following functions:

- `KosTaskInitEx()`
- `KosTaskInit()`
- `KosTaskInitFromSegEx()`
- `KosTaskInitFromSeg()`
- `KosTaskLaunch()`

Using the `params` parameter, these functions receive parameters of the created process via a structure containing the following fields:

- `eiid` – pointer to the process class name.
- `endpointsCount` – number of provided endpoints.

  The field can have a zero value if the process does not provide endpoints.
- `endpoints` – pointer to the array of structures containing the characteristics of provided endpoints (names and RIIDs of endpoints, names of interfaces).

  The type of structure is defined in the header file `sysroot-*-kos/include/services/handle/if_connection.h` from the KasperskyOS SDK.

  The field can have the `RTL_NULL` value if the process does not provide endpoints.
- `args` – pointer to the array of program startup parameters.

  The `RTL_NULL` value must be the last element of the array.
- `envs` – pointer to the array of environment variables of the program.

  The `RTL_NULL` value must be the last element of the array.
- `flags`:

- `KOS_TASK_FLAGS_NONE` – no flags.
- `KOS_TASK_FLAG_DUMPABLE` – the process switches to the "frozen" state as a result of an unhandled exception.

  For details about the "frozen" state, see Managing processes (low-level API task_api.h).

- `componentTree` – pointer to the structure containing information from the formal specification of the solution component.

  The type of structure is defined in the header file `sysroot-*-kos/include/nk/types.h` from the KasperskyOS SDK. This structure is an element of automatically generated transport code.

  If the field has a value other than `RTL_NULL`, the values in the `eiid`, `endpointsCount` and `endpoints` fields will be ignored, and the process class name and parameters of the provided endpoints (including the parameters of endpoints of embedded components) will be taken from the formal specification of the solution component.

The `KosTaskInitEx()`, `KosTaskInit()`, `KosTaskInitFromSegEx()` and `KosTaskInitFromSeg()` functions use the `outTask` output parameter to pass the pointer to the address of the object describing the child process. This object is a structure that is created in the memory of the parent process and the child process. The developer of a KasperskyOS-based solution does not need to perform operations with the fields of this structure. However, the pointer to this structure must be used as the process ID when calling API functions. A child process can get the address of the object describing it by calling the `KosTaskGetSelf()` function.

If statically created IPC channels are used for access to endpoints provided by a server process, the object describing this server process must be linked to the structures containing information about endpoints from the formal specification of the solution component. This is necessary so that client processes can receive information about endpoints provided by the server process when creating a static IPC channel. To link an object describing a child server process to structures containing information about endpoints from the formal specification of the solution component, this information must be passed through the `componentTree` field of the `params` parameter when calling the `KosTaskInit*()` functions or `KosTaskLaunch()` function. A server process that is already running can link the object describing it to structures containing information about endpoints from the formal specification of the solution component by calling the `KosTaskSetComponentTree()` function. This is required if the running server process does not have a parent process.

When the `KosTaskInitEx()`, `KosTaskInit()` or `KosTaskLaunch()` function is called, the ELF image from the defined executable file in ROMFS is loaded into the memory of the created process. If the ELF image contains the symbol table `.symtab` and string table `.strtab`, they are loaded into process memory. Using these tables, the kernel receives the names of functions for generating stack backtrace data (call stack information).

To get information about the ELF image loaded into process memory, call the `KosTaskGetElfSegs()` function.

When the `KosTaskInit()` or `KosTaskLaunch()` function is called, one of the following values is used as the process name and executable file name:

- Value of the `eiid` field, if the value of the `componentTree` field is equal to `RTL_NULL`.
- Name of the process class from the formal specification of the solution component, if the `componentTree` field value is different from `RTL_NULL`.

Likewise, these values are applied as the process name and/or executable file name if you call the `KosTaskInitEx()` or `KosTaskInitFromSegEx()` function with the `RTL_NULL` value in the `name` parameter and/or the `path` parameter. These values are also applied as the process name if you call the `KosTaskInitFromSeg()` function with the `RTL_NULL` value in the `name` parameter.

To use the `KosTaskInitFromSegEx()` and `KosTaskInitFromSeg()` functions, MDL buffers containing the ELF image segments must be created in advance. You must use the `segs` parameter to define the ELF image segments to be loaded into the memory of the created process.

You must use the `entry` and `relocBase` parameters of the `KosTaskInitFromSegEx()` function to define the program entry point and the ELF image load offset, respectively. The program entry point is the sum of the address specified in the `e_entry` field of the ELF image header and the ELF image load offset. The `KnElfCreateVmSegEx()` function declared in the header file `sysroot-*-kos/include/coresrv/elf/elf_api.h` from the KasperskyOS SDK generates a random offset for loading the ELF image and calculates the address of the program entry point according to this offset. (The ELF image load offset must be a random value to ensure ASLR support. For details about ASLR, see Managing processes (low-level API task_api.h).)

You can use the `KosTaskInitFromSegEx()` function to load the symbol table `.symtab`, string table `.strtab` and ELF image header into the memory of the created process. The ELF image header should be loaded if data from this header must be available in the created process.

Data passed to the `KosTaskInitFromSegEx()` function via the `segs`, `entry`, and `relocBase` parameters and the parameters associated with loading the symbol table `.symtab` and string table `.strtab` are prepared by the `KnElfCreateVmSegEx()` function declared in the header file `sysroot-*-kos/include/coresrv/elf/elf_api.h` from the KasperskyOS SDK.

The `KosTaskInitFromSeg()` function is a simplified version of the `KosTaskInitFromSegEx()` function and does not let you load the symbol table `.symtab`, string table `.strtab` and ELF image header into process memory, and does not let you define the ELF image load offset (instead, it sets a null offset).

The `KosTaskLaunch()` function creates and immediately starts a process without the capability to statically create IPC channels.

Statically created IPC channels

Before starting processes, you can create IPC channels between them. You can create multiple IPC channels with different names between one client process and one server process. You can create IPC channels with the same name between one server process and multiple client processes.

To create an IPC channel with a name matching the name of a server process class, call the `KosTaskConnect()` function.

To create an IPC channel with a defined name, call the `KosTaskConnectToService()` function.

To use the created IPC channel, you need to get the client IPC handle on the client process side by calling the `ServiceLocatorConnect()` function. On the server process side, you need to get the server IPC handle by calling the `ServiceLocatorRegister()` function. These functions use the `channelName` parameter to receive the name of the IPC channel. (The `ServiceLocatorConnect()` and `ServiceLocatorRegister()` functions are declared in the header file `sysroot-*-kos/include/coresrv/sl/sl_api.h` from the KasperskyOS SDK.)

Transferring handles

A parent process can transfer one or more handles to a child process that is not yet running. (General information about transferring handles is provided in the Transferring handles section.)

To pass a handle to a child process, call the `KosTaskTransferResource()` function while specifying the handle of the resource transfer context object, the permissions mask, and the conditional name of the descendant of the transferred handle in addition to the other parameters.

To find the descendant of a handle transferred by a parent process, call the `KosTaskLookupResource()` function while specifying the conditional name of the descendant of the handle that was defined by the parent process when calling the `KosTaskTransferResource()` function.

Starting processes

To create and immediately start a process without statically creating IPC channels, call the `KosTaskLaunch()` function.

To start an already created process for which you can create the necessary IPC channels before starting this process, call the `KosTaskRunEx()` or `KosTaskRun()` function.

Use the `fsBackend` parameter of the `KosTaskRunEx()` function to specify whether to use the kernel or the system program `fsusr` to support the ROMFS file system for the started process. Use of the `fsusr` program ensures that the ROMFS image resides in the user space. The user space can host a significantly larger ROMFS image than the kernel space.

To enable the use of a ROMFS file system residing in the user space, include the `fsusr` program into the KasperskyOS-based solution and create an IPC channel named `kl.core.FSUsr` from the process that needs to use this file system to the process named `kl.core.FSUsr`. (The client portion of the `fsusr` program is included in the `libkos` library.) To check whether the `fsusr` program is included in your KasperskyOS SDK, verify that the `sysroot-*-kos/bin/fsusr` executable file is available.

To specify how to support the file system in an already running process, call the `KosTaskSetSelfFSBackend()` function. This function can be used as follows. The parent process indicates that the `fsusr` program supports its ROMFS file system, and loads the required ROMFS image by calling the `KnFsChange()` function. (The `KnFsChange()` function is declared in the header file `sysroot-*-kos/include/coresrv/fs/fs_api.h` from the KasperskyOS SDK.) Then the parent process starts the child process by calling the `KosTaskRunEx()` function and specifies that the ROMFS file system for the child process is supported by the `fsusr` program. As a result, the child process will use the ROMFS image that is placed in the user space by the parent process by default.

If the parent process does not need to terminate the child process or wait for its termination, the object describing the child process must be deleted and the counter for links to it must be reset by using the `KosTaskPut()` function after the child process is started. The `KosTaskLaunch()` function calls the `KosTaskPut()` function after the child process is started.

Terminating processes

The API terminates and waits for termination of child processes.

To terminate a child process, call the `KosTaskStop()` or `KosTaskStopAndWait()` function.

To wait for a child process to terminate on its own initiative, call the `KosTaskWait()` function.

To ensure that the kernel object describing a child process is deleted after this process is terminated, its handle in the parent process must be closed before or after termination of the child process. The handle of the child process is closed when the object describing the child process is deleted from the memory of the parent process. To delete an object describing a child process, reset the counter for links to this object by using the `KosTaskPut()` function. The `KosTaskLaunch()` function calls the `KosTaskPut()` function after the child process is started.

For details about terminating processes, see Managing processes (low-level API task_api.h).

Information about API functions

task.h functions

Function	Information about the function
	Purpose

`KosTaskInitEx()`	Creates a process. Parameters • [in] `params` – pointer to the structure containing process parameters. • [in] `name` – pointer to the process name. • [in] `path` – pointer to the name of the executable file in ROMFS. • [out] `outTask` – pointer to the address of the object describing the process. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosTaskInit()`	Purpose Creates a process. Parameters • [in] `params` – pointer to the structure containing process parameters. • [out] `outTask` – pointer to the address of the object describing the process. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosTaskInitFromSegEx()`	Purpose Creates a process. Parameters • [in] `params` – pointer to the structure containing process parameters. • [in] `name` – pointer to the process name. • [in] `path` – pointer to the name of the executable file in ROMFS. • [in] `count` – number of loaded ELF image segments. • [in] `relocBase` – ELF image load offset (in bytes). • [in] `entry` – address of the program entry point. • [in,out] `segs` – pointer to the array of structures describing ELF image segments. The type of structure is defined in the header file `sysroot-*-kos/include/coresrv/vmm/vmm_types.h` from the KasperskyOS SDK. If the `loadAddr` field of this structure is set equal to `0`, the load address of the ELF image segment will be selected automatically and written to the `loadAddr` field. • [in] `symTabIndex` – index of the ELF image segment containing the symbol table `.symtab`. Enumeration starts with zero. • [in] `symTabSize` – size of the symbol table `.symtab` (in bytes). • [in] `strTabIndex` – index of the ELF image segment containing the string table `.strtab`. Enumeration starts with zero. • [in] `strTabSize` – size of the string table `.strtab` (in bytes). • [in] `elfHdrData` – pointer to the buffer containing the ELF image header. • [in] `elfHdrSize` – size of the ELF image header (in bytes). • [out] `outTask` – pointer to the address of the object describing the process. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

`KosTaskInitFromSeg()`	Purpose Creates a process. Parameters • [in] `params` – pointer to the structure containing process parameters. • [in] `name` – pointer to the process name. • [in] `count` – number of loaded ELF image segments. • [in] `entry` – address of the program entry point. • [in,out] `segs` – pointer to the array of structures describing ELF image segments. The type of structure is defined in the header file `sysroot-*-kos/include/coresrv/vmm/vmm_types.h` from the KasperskyOS SDK. If the `loadAddr` field of this structure is set equal to `0`, the load address of the ELF image segment will be selected automatically and written to the `loadAddr` field. • [out] `outTask` – pointer to the address of the object describing the process. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosTaskGetElfSegs()`	Purpose Gets information about the ELF image loaded in process memory. Parameters • [in] `task` – pointer to the object that describes the process. • [out] `segsBuf` – pointer to the array used to save structures describing ELF image segments. The type of structure is defined in the header file `sysroot-*-kos/include/coresrv/vmm/vmm_types.h` from the KasperskyOS SDK. • [in] `segsBufSize` – size (in bytes) of the buffer used to store the array of structures that describe the ELF image segments. • [out] `segsCount` – pointer to the number of loaded ELF image segments. • [out] `symTabIndex` – pointer to the index of the ELF image segment containing the symbol table `.symtab`. Enumeration starts with zero. • [out] `symTabSize` – pointer to the size of the symbol table `.symtab` (in bytes). • [out] `strTabIndex` – pointer to the index of the ELF image segment containing the string table `.strtab`. Enumeration starts with zero. • [out] `strTabSize` – pointer to the size of the string table `.strtab` (in bytes). • [out] `hdrBuf` – pointer to the buffer used to store the ELF image header. • [in] `hdrBufSize` – size (in bytes) of the buffer used to store the ELF image header. • [out] `hdrSize` – size (in bytes) of the ELF image header. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosTaskConnect()`	Purpose Creates an IPC channel. Parameters • [in] `cl` – pointer to the object that describes the client process. • [in] `sr` – pointer to the object that describes the server process. Returned values

	If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosTaskConnectToService()`	Purpose Creates an IPC channel with the defined name. Parameters • [in] `cl` – pointer to the object that describes the client process. • [in] `sr` – pointer to the object that describes the server process. • [in] `name` – pointer to the name of the IPC channel. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosTaskSetArgs()`	Purpose Defines the program startup parameters. Parameters • [in] `task` – pointer to the object that describes the process. • [in] `args` – pointer to the array of program startup parameters. The `RTL_NULL` value must be the last element of the array. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information The function is used by the `libkos` library when creating processes.
`KosTaskGetArgsList()`	Purpose Gets the program startup parameters. Parameters • [in] `task` – pointer to the object that describes the process. • [out] `args` – pointer to the address of the array of program startup parameters, or `RTL_NULL` if these parameters are not available. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information To delete an array of program startup parameters, call the `KosTaskFreeArgsList()` function.
`KosTaskFreeArgsList()`	Purpose Deletes the array of program startup parameters that was received from the `KosTaskGetArgsList()` call. Parameters • [in] `args` – pointer to the array of program startup parameters. Returned values N/A
`KosTaskSetEnv()`	Purpose Defines environment variables of a program. Parameters

	• [in] `task` – pointer to the object that describes the process. • [in] `envs` – pointer to the array of environment variables of the program. The `RTL_NULL` value must be the last element of the array. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information The function is used by the `libkos` library when creating processes.
`KosTaskGetEnvList()`	Purpose Gets the environment variables of a program. Parameters • [in] `task` – pointer to the object that describes the process. • [out] `envs` – pointer to the address of the array of program environment variables, or `RTL_NULL` if these variables are not available. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information To delete an array of program environment variables, call the `KosTaskFreeEnvList()` function.
`KosTaskFreeEnvList()`	Purpose Deletes the array of program environment variables that was received from calling the `KosTaskGetEnvList()` function. Parameters • [in] `envs` – pointer to the array of environment variables of the program. Returned values N/A
`KosTaskRunEx()`	Purpose Starts a process. Parameters • [in] `task` – pointer to the object that describes the process. • [in] `fsBackend` – parameter that specifies whether the ROMFS file system for the started process is supported by the kernel or by a system program. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosTaskRun()`	Purpose Starts a process. Parameters • [in] `task` – pointer to the object that describes the process. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosTaskStop()`	Purpose

	Terminates a process. Parameters • [in] `task` – pointer to the object that describes the process. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosTaskStopAndWait()`	Purpose Terminates a process and waits for the termination of this process. Parameters • [in] `task` – pointer to the object that describes the process. • [in] `timeout` – timeout in milliseconds, or `INFINITE_TIMEOUT` to define an unlimited timeout. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosTaskWait()`	Purpose Waits for the termination of a process. Parameters • [in] `task` – pointer to the object that describes the process. • [in] `timeout` – timeout in milliseconds, or `INFINITE_TIMEOUT` to define an unlimited timeout. • [out] `retcode` – pointer to the process exit code. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosTaskGetHandle()`	Purpose Gets the process handle. Parameters • [in] `task` – pointer to the object that describes the process. Returned values Process handle.
`KosTaskLaunch()`	Purpose Creates and starts a process. Parameters • [in] `params` – pointer to the structure containing process parameters. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosTaskGetSelf()`	Purpose Gets the address of an object that describes a calling process. Parameters N/A Returned values

	Pointer to the object describing the process, or `RTL_NULL` if this object was deleted. Additional information The function adds 1 to the counter for links to the object describing the process, therefore the `KosTaskPut()` function must be called after this function is called.
`KosTaskPut()`	Purpose Subtracts 1 from the counter for links to the object describing the process. Parameters • [in] `task` – pointer to the object that describes the process. Returned values N/A
`KosTaskAddEndpoints()`	Purpose Registers endpoints. Parameters • [in,out] `task` – pointer to the object that describes the process. • [in] `endpoints` – pointer to the array of structures describing endpoints. The type of structure is defined in the header file `sysroot-*-kos/include/services/handle/if_connection.h` from the KasperskyOS SDK. • [in] `count` – number of endpoints. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information The function is used by the `libkos` library when creating processes.
`KosTaskSetSelfFSBackend()`	Purpose Defines whether the ROMFS file system for a calling process is supported by the kernel or by a system program. Parameters • [in] `fsBackend` – parameter that defines whether the ROMFS file system is supported by the kernel or by a system program. Returned values N/A
`KosTaskGetSelfFSBackend()`	Purpose Gets information indicating whether the ROMFS file system for a calling process is supported by the kernel or by a system program. Parameters N/A Returned values Value indicating whether the ROMFS file system is supported by the kernel or by a system program.
`KosTaskSetComponentTree()`	Purpose Creates structures containing information about endpoints from the formal specification of the solution component and links these structures to the object describing the process. Parameters

	• [in] `task` – pointer to the object that describes the process. • [in] `root` – pointer to the structure containing information from the formal specification of the solution component. The type of structure is defined in the header file `sysroot-*-kos/include/nk/types.h` from the KasperskyOS SDK. This structure is an element of automatically generated transport code. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosTaskTransferResource()`	Purpose Transfers a handle to a process that is not yet running. Parameters • [in] `task` – pointer to the object that describes the process. • [in] `srcHandle` – transferred handle. • [in] `srcBadge` – handle of the resource transfer context object. • [in] `dstRights` – permissions mask of the descendant of the transferred handle. • [in] `handleName` – name of the descendant of the transferred handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosTaskLookupResource()`	Purpose Searches for a handle based on the name defined by the `KosTaskTransferResource()` function. Parameters • [in] `handleName` – name of the handle. Returned values If successful, it returns the handle. Otherwise, it returns `INVALID_HANDLE`.

Managing processes (low-level API task_api.h)

The API is defined in the header file `sysroot-*-kos/include/coresrv/task/task_api.h` from the KasperskyOS SDK.

Main capabilities of the API:

- Create, start, and terminate processes.
- Handle exceptions.
- Get information about processes, including information about the reasons for their termination.
- Define priorities and scheduler classes for initial threads of processes.

Information about API functions is provided in the table below.

> The `libkos` library also provides a high-level API for process management. This API is defined in the header file `sysroot-*-kos/include/kos/task.h` from the KasperskyOS SDK. It is recommended to use this specific API. The low-level API should be used only if the high-level API does not have sufficient capabilities.

Creating and starting processes

To create a process, call the `KnTaskCreate()` or `KnTaskCreateEx()` function. These functions create an *"empty" process*, which is a process in whose memory the ELF image of the program is not loaded. Before starting this process, complete the following steps:

1. Create a seed value by calling the `KosRandomGenerate()` function, which is declared in the header file `sysroot-*-kos/include/kos/random/random_api.h` from the KasperskyOS SDK.

   This step is required for completion of the next step.

2. Define the seed value by calling the `KnTaskReseedAslr()` function.

   This step is required for address space layout randomization. *Address Space Layout Randomization* (ASLR) is the use of random addresses for the location of data structures (ELF image, dynamic libraries, stack and heap) in process memory to make it harder to exploit vulnerabilities associated with a conventional process address space structure that is known by a hacker in advance.

   The `KnTaskReseedAslr()` function defines the seed value of the random number generator that is used to automatically select the base address of an allocated virtual memory region in functions such as `KnVmAllocate()`, `KnPmmMdlMap()`, `KnIoDmaMap()`, and `KnTaskVmReserve()`. The stack and heap are created in the process and dynamic libraries are loaded into the process memory by the operating system using the `KnVmAllocate()` function. The `addr` parameter is set to zero so that the address of the allocated virtual memory region is selected automatically (as a random value).

3. Wipe the seed value created at step 1 from memory.

   This step is required for security purposes. To complete this step, use the `RtlRandomMemSanitize()` function, which is declared in the header file `sysroot-*-kos/include/rtl/random.h` from the KasperskyOS SDK.

4. Load ELF image segments into process memory by using the `KnTaskLoadSeg()` function.

   In the `loadAddr` field of the `seg` parameter, specify the load address of the ELF image segment. To ensure ASLR support (as a supplement to step 2), the load address of the ELF image segment specified in the ELF file must be increased by the ELF image load offset. The ELF image load offset must be a random value. The `KnElfCreateVmSegEx()` function declared in the header file `sysroot-*-kos/include/coresrv/elf/elf_api.h` from the KasperskyOS SDK generates the random offset for loading the ELF image and calculates the load addresses of ELF image segments according to this offset.

   As a result of this step, MDL buffers that contain ELF image segments will be mapped to virtual memory of the process.

5. [Optional] Load the symbol table `.symtab` and string table `.strtab` into process memory by calling the `KnTaskLoadElfSyms()` function.

   The load addresses of ELF image segments containing the symbol table `.symtab` and string table `.strtab` must be calculated just like the load addresses of other ELF image segments. This calculation is performed by the `KnElfCreateVmSegEx()` function declared in the header file `sysroot-*-kos/include/coresrv/elf/elf_api.h` from the KasperskyOS SDK.

   The kernel uses the symbol table `.symtab` and string table `.strtab` to get the names of functions for generating stack backtrace data (call stack information).

6. Define the program entry point and the ELF image load offset by calling the `KnTaskSetInitialState()` function.

The program entry point is the sum of the address specified in the `e_entry` field of the ELF image header and the ELF image load offset. The `KnElfCreateVmSegEx()` function declared in the header file `sysroot-*-kos/include/coresrv/elf/elf_api.h` from the KasperskyOS SDK generates a random offset for loading the ELF image and calculates the address of the program entry point according to this offset.

7. [Optional] Load the ELF image header into process memory by calling the `KnTaskSetElfHdr()` function.

   This step must be completed if data from the ELF image header must be available in the created process.

The handle of a process can be transferred to another process via IPC.

By default, the initial thread of a process is a standard thread whose priority can take values ranging from 0 to 15. (For details about thread scheduler classes, see Managing threads (low-level API thread_api.h).) To change the scheduler class and/or priority of the initial thread of a process, call the `KnTaskSetInitialPolicy()` function. To change the priority of the initial thread of a process, call the `KnTaskSetInitialThreadPriority()` function. The `KnTaskSetInitialPolicy()` and `KnTaskSetInitialThreadPriority()` functions can be used after the program entry point is defined. These functions can also be used after starting the process.

To start the process, call the `KnTaskResume()` function. A running process cannot be stopped.

Before a process is started, it receives data from its parent process via a static connection page. A *static connection page* (SCP) is a set of structures containing data for statically creating IPC channels, startup parameters, and environment variables of a program. A parent process writes data to the SCP of a child process by calling the `KnTaskSetEnv()` function. When a child process is started, it reads data from the SCP by calling the `KnTaskGetEnv()` function, then it deletes the SCP by calling the `KnTaskFreeEnv()` function. All three functions do not need to be explicitly called because their calls are made by the `libkos` library.

Terminating processes

Process termination includes the following:

- Terminating all threads of the process.
- Freeing the memory of the process.
- Freeing system resources and user resources that are exclusively owned by the process.

  When a process is terminated, all the handles that it owns are closed. If a closed handle was the only handle of a resource, this resource is freed.

A process can be terminated for the following reasons:

- On its own initiative.

  The `KnTaskExit()` function is called, or all threads of the process are terminated.

- By external request.

  The `KnTaskTerminate()` function is called.

- As a result of an unhandled exception (crash).

  By calling the `KnTaskPanic()` function, a process can purposefully initiate an exception that cannot be handled and leads to the process terminating. This assures that the process can terminate itself. (For example, if the process contains threads attached to interrupts, the `KnTaskExit()` cannot terminate the process, while the `KnTaskPanic()` function can.)

The exit codes of processes are defined by the developer of the KasperskyOS-based solution. These codes must be specified in the `status` parameter of the `KnTaskExit()` function. If a process was terminated due to the termination of all its threads, the exit code of this process will be the exit code of its initial thread. To get the exit code of a process that was terminated on its own initiative, call the `KnTaskGetExitCode()` function.

To get information regarding the reason for process termination, call the `KnTaskGetExitStatus()` function. This function determines whether a process was terminated on its own initiative, by external request, or unexpectedly.

To get information about an unhandled exception that led to an unexpected termination of a process, call the `KnTaskGetExceptionInfo()` function. This information includes the exception code and the context of the thread in which this exception occurred.

If the process was created by calling the `KnTaskCreateEx()` function with the `TaskExceptionFreezesTask` flag in the `flags` parameter, an unhandled exception will cause this process to switch to a "frozen" state instead of terminating. When a process is switched to a *"frozen" state*, its threads are terminated as a result of the unhandled exception but its resources are not freed, which means that you can get information about this process. To get the context of a thread that is part of a frozen process, call the `KnTaskGetThreadContext()` function. To get information about the virtual memory region that belongs to a frozen process, call the `KnTaskGetNextVmRegion()` function. This information includes the base address and size of the virtual memory region, and the access rights to this virtual memory region. Before a process switches to the frozen state, stack backtrace data (call stack information) is printed for the thread in which the unhandled exception occurred. To terminate a frozen process, call the `KnTaskTerminateAfterFreezing()` function.

To ensure that the kernel object describing a process is deleted after this process is terminated, each of its handles must be closed or revoked in all processes that own these handles before or after termination of this process. To do so, use the `KnHandleClose()` and/or `KnHandleRevoke()` functions that are declared in the header file `sysroot-*-kos/include/coresrv/handle/handle_api.h` from the KasperskyOS SDK.

Handling exceptions

To register an exception handling function, call the `KnTaskSetExceptionHandler()` function. This function deregisters the previous exception handling function and returns its ID. By saving this ID, you can subsequently register the previous exception handling function again.

An exception handling function is called when an exception occurs in any process thread. If the exception is successfully handled, this function returns a value other than zero. Otherwise it returns zero. The input parameter of the exception handling function is a structure containing information about the exception. The type of this structure is defined in the header file `sysroot-*-kos/include/thread/tcbpage.h` from the KasperskyOS SDK.

If an exception handling function registered at the process level failed to successfully handle an exception, the exception handling function registered at the level of the thread in which the exception occurred will be called. (For details about handling exceptions at the thread level, see Managing threads (low-level API thread_api.h)).

Reserving memory in a child process

The API includes the `KnTaskVmReserve()` and `KnTaskVmFree()` functions, which reserve and free virtual memory regions, respectively, in a child process that does not yet have a defined program entry point. These functions comprise only a part of the functionality intended to increase control of the parent process over the virtual address space of a child process. This functionality is currently under development.

Getting the GSI address

Global system information (GSI) is a structure containing system information, such as the timer count since the kernel started, the timer counts per second, the number of processors (processor cores) in active state, and processor cache data. The type of structure is defined in the header file `sysroot-*-kos/include/task/pcbpage.h` from the KasperskyOS SDK. The `KnTaskGetGsi()` function gets the GSI address. This function is intended for use by the `libc` library.

Getting the PCB address

A *process control block* (PCB) is a structure containing process information that is used by the kernel to manage this process. The type of structure is defined in the header file `sysroot-*-kos/include/task/pcbpage.h` from the KasperskyOS SDK. The `KnTaskGetPcb()` function gets the PCB address. This function is intended for use by the `libc` library.

Information about API functions

task_api.h functions

Function	Information about the function
`KnTaskCreate()`	Purpose Creates a process. Parameters • [in] `name` – pointer to the process name. • [in] `eiid` – pointer to the process class name. • [in] `path` – parameter that must have the value `RTL_NULL`. • [in,optional] `stackSize` – thread stack limit (in bytes) used by default when creating process threads. If `0` is specified, the stack will be 1 MB by default. • [in] `priority` – priority of the initial thread. The parameter type is defined in the header file `sysroot-*-kos/include/coresrv/thread/thread_api.h` from the KasperskyOS SDK. • [out] `outHandle` – pointer to the process handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnTaskCreateEx()`	Purpose Creates a process. Parameters • [in] `name` – pointer to the process name. • [in] `eiid` – pointer to the process class name. • [in] `path` – parameter that must have the value `RTL_NULL`. • [in,optional] `stackSize` – thread stack limit (in bytes) used by default when creating process threads. If `0` is specified, the stack will be 1 MB by default. • [in] `priority` – priority of the initial thread. The parameter type is defined in the header file `sysroot-*-kos/include/coresrv/thread/thread_api.h` from the KasperskyOS SDK. • [in] `flags` – flags defining the parameters for creating the process. • [out] `outHandle` – pointer to the process handle. Returned values

	If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information In the `flags` parameter, you can specify the following flags: • `TaskExceptionTerminatesTask` – the process is terminated as a result of an unhandled exception. • `TaskExceptionFreezesTask` – the process switches to the "frozen" state as a result of an unhandled exception. This flag cannot be specified together with the `TaskExceptionTerminatesTask` flag. • `TaskEmpty` – creates an "empty" process. This mandatory flag must be set.
`KnTaskGetGsi()`	Purpose Gets the GSI address for the calling process. Parameters N/A Returned values Pointer to the GSI. The data type for GSI storage is defined in the header file `sysroot-*-kos/include/task/pcbpage.h` from the KasperskyOS SDK.
`KnTaskGetPcb()`	Purpose Gets the address of the process control block (PCB) for the calling process. Parameters N/A Returned values Pointer to the PCB. The data type for PCB storage is defined in the header file `sysroot-*-kos/include/task/pcbpage.h` from the KasperskyOS SDK.
`KnTaskGetEnv()`	Purpose Gets the SCP address of the calling process. Parameters • [out] `envSize` – pointer to the size of the SCP (in bytes). Returned values Pointer to the SCP, or `RTL_NULL` if the process does not have an SCP.
`KnTaskSetEnv()`	Purpose Writes data to the SCP of a process. Parameters • [in] `task` – process handle. • [in] `env` – pointer to the buffer containing data to be written to the SCP. • [in] `envSize` – size (in bytes) of the data to be written to the SCP. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnTaskFreeEnv()`	Purpose Deletes the SCP of the calling process.

	Parameters N/A Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnTaskSetElfHdr()`	Purpose Writes the ELF image header to the PCB of a process. Parameters • [in] `h` – process handle. • [in] `hdrData` – pointer to the buffer containing the ELF image header. • [in] `hdrSize` – size (in bytes) of the ELF image header. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnTaskResume()`	Purpose Starts a process. Parameters • [in] `task` – process handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnTaskExit()`	Purpose Terminates the calling process. Parameters • [in] `status` – exit code of the process. Returned values Error code. Additional information Does not terminate the process as long as it contains threads attached to interrupts.
`KnTaskTerminate()`	Purpose Terminates a process. Parameters • [in] `task` – process handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnTaskGetExitStatus()`	Purpose Gets information about the reason for process termination. Parameters • [in] `task` – handle of the terminated process. • [out] `status` – pointer to the value indicating the reason for process termination. The data type for storing this value is defined in the header file `sysroot-*-`

	`kos/include/task/pcbpage.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information You can use the `status` parameters to get the following values: • `TaskExitUnexpected` – the process terminated unexpectedly. • `TaskExitNormal` – the process terminated on its own initiative. • `TaskExitTerminated` – the process was terminated by an external request.
`KnTaskGetExceptionInfo()`	Purpose Gets information about an unhandled exception that led to an unexpected termination of a process. Parameters • [in] `task` – handle of the unexpectedly terminated process. • [out] `excType` – pointer to an exception code that is non-specific for any processor architecture. The data type for storing this code is defined in the header file `sysroot-*-kos/include/hal/exc_codes.h` from the KasperskyOS SDK. • [out] `excNo` – pointer to an exception code that is specific to the utilized processor architecture. The data type for storing this code is defined in the header file `sysroot-*-kos/include/hal/exc_codes.h` from the KasperskyOS SDK. • [out] `exceptionContext` – pointer to the structure containing the context of the thread in which the exception occurred. The type of structure is defined in the header file `sysroot-*-kos/include/hal/*/frame.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnTaskGetThreadContext()`	Purpose Gets the context of a thread that is part of a frozen process. Parameters • [in] `task` – handle of the frozen process. • [in] `index` – thread index. It is used to enumerate threads. Enumeration starts with zero. A thread in which an unhandled exception occurred has a zero index. • [out] `context` – pointer to the structure containing the thread ID (TID) and context of the thread. The type of structure is defined in the header file `sysroot-*-kos/include/thread/context.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnTaskGetNextVmRegion()`	Purpose Gets information about the virtual memory region that belongs to a frozen process. Parameters • [in] `task` – handle of the frozen process. • [in] `after` – address that is followed by the virtual memory region. • [out] `next` – pointer to the base address of the virtual memory region. • [out] `size` – pointer to the size of the virtual memory region (in bytes).

	• [out] `flags` – pointer to the flags indicating the parameters of the virtual memory region. The flags are defined in the header file `sysroot-*-kos/include/vmm/flags.h` from the KasperskyOS SDK. • [out] `outHandle` – pointer to the handle of the MDL buffer mapped to the virtual memory region. The `KnTaskGetNextVmRegion()` function creates an MDL buffer from physical memory mapped to a virtual memory region (even if there is already an MDL buffer mapped to this virtual memory region). Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnTaskTerminateAfterFreezing()`	Purpose Terminates a frozen process. Parameters • [in] `task` – handle of the frozen process. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnTaskGetExitCode()`	Purpose Gets the exit code of a process that terminated on its own initiative. Parameters • [in] `task` – handle of the terminated process. • [out] `exitCode` – pointer to the process exit code. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnTaskGetId()`	Purpose Gets the process ID (PID) for the calling process. Parameters N/A Returned values Process ID. The ID type is defined in the header file `sysroot-*-kos/include/task/pidtype.h` from the KasperskyOS SDK.
`KnTaskGetName()`	Purpose Gets the name of a calling process. Parameters • [out] `name` – pointer to the buffer used to store the process name. • [in] `msize` – size of the buffer used to store the process name (in bytes). Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnTaskGetPath()`	Purpose Gets the name of the executable file (in ROMFS) that was used to create the calling process. Parameters • [out] `path` – pointer to the buffer used to store the executable file name. • [in] `msize` – size (in bytes) of the buffer used to store the name of the executable file.

	Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnTaskGetInitialThreadPriority()`	Purpose Gets the priority of the initial thread of a process. Parameters • [in] `task` – process handle. • [out] `priority` – pointer to the priority of the initial thread of the process. The data type for storing the priority of a thread is defined in the header file `sysroot-*-kos/include/coresrv/thread/thread_api.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnTaskSetInitialThreadPriority()`	Purpose Defines the priority of the initial thread of a process. Parameters • [in] `task` – process handle. • [in] `priority` – priority of the initial thread of a process. The parameter type is defined in the header file `sysroot-*-kos/include/coresrv/thread/thread_api.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnTaskSetExceptionHandler()`	Purpose Registers the exception handling function for the calling process. Parameters • [in,optional] `handler` – ID of the exception handling function, or `RTL_NULL` to deregister the previously registered function without registering a new one. The parameter type is defined in the header file `sysroot-*-kos/include/task/pcbpage.h` from the KasperskyOS SDK. Returned values ID of the previously registered exception handling function if one exists, otherwise `RTL_NULL`. The type of returned value is defined in the header file `sysroot-*-kos/include/task/pcbpage.h` from the KasperskyOS SDK.
`KnTaskLoadSeg()`	Purpose Loads an ELF image segment into process memory. Parameters • [in] `h` – process handle. • [in] `seg` – pointer to the structure describing the ELF image segment. If the `loadAddr` field of this structure is set equal to `0`, the ELF image segment load address will be selected automatically. The type of structure is defined in the header file `sysroot-*-kos/include/coresrv/vmm/vmm_types.h` from the KasperskyOS SDK. • [out] `actual` – pointer to the load address of the ELF image segment. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnTaskVmReserve()`	Purpose Reserves a virtual memory region in a process.

	Parameters • [in] `task` – process handle. • [in,optional] `hint` – page-aligned, preferred base address of the virtual memory region, or `RTL_NULL` to select this address automatically. • [in] `size` – size of the virtual memory region in bytes. • [in] `flags` – flags defining the parameters of the virtual memory region. The flags are defined in the header file `sysroot-*-kos/include/vmm/flags.h` from the KasperskyOS SDK. • [out] `addr` – pointer to the base address of the reserved virtual memory region. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information In the `flags` parameter, you can specify the following flags: • `VMM_FLAG_LOCKED` – commits the virtual memory region while allocating the entire required volume of physical memory. When the `KnTaskVmReserve()` function is called with this flag, the memory is not committed. This flag affects subsequent commitment-related actions taken on the reserved virtual memory region. • `VMM_FLAG_READ`, `VMM_FLAG_WRITE`, `VMM_FLAG_EXECUTE` and `VMM_FLAG_RWX_MASK` – flags defining the access rights to the virtual memory region. • `VMM_FLAG_LOW_GUARD`, `VMM_FLAG_HIGH_GUARD` – adds a protected page to the beginning and end of the virtual memory region, respectively. The size of the protected page is not included in the size of the virtual memory region. Permissible combinations of flags defining the access rights to the virtual memory region: • `VMM_FLAG_READ` – read access. • `VMM_FLAG_READ | VMM_FLAG_WRITE` – read-and-write access. • `VMM_FLAG_READ | VMM_FLAG_EXECUTE` – read-and-execute access. • `VMM_FLAG_RWX_MASK` or `VMM_FLAG_READ | VMM_FLAG_WRITE | VMM_FLAG_EXECUTE` – read/write/execute access.
`KnTaskVmFree()`	Purpose Frees the virtual memory region that was reserved by calling the `KnTaskVmReserve()` function. Parameters • [in] `task` – process handle. • [in] `base` – page-aligned base address of the virtual memory region. • [in] `size` – size of the virtual memory region in bytes. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnTaskSetInitialState()`	Purpose Defines the program entry point and the ELF image load offset. Parameters • [in] `h` – process handle. • [in] `state` – pointer to the structure containing the address of the program entry point and the ELF image load offset (in bytes).

	Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnTaskLoadElfSyms()`	Purpose Loads the symbol table `.symtab` and string table `.strtab` into process memory. Parameters • [in] `h` – process handle. • [in] `symTabSeg` – pointer to the structure describing the ELF image segment with the symbol table `.symtab`. The type of structure is defined in the header file `sysroot-*-kos/include/coresrv/vmm/vmm_types.h` from the KasperskyOS SDK. • [in] `symTabSize` – size of the symbol table `.symtab` (in bytes). • [in] `strTabSeg` – pointer to the structure describing the ELF image segment with the string table `.strtab`. The type of structure is defined in the header file `sysroot-*-kos/include/coresrv/vmm/vmm_types.h` from the KasperskyOS SDK. • [in] `strTabSize` – size of the string table `.strtab` (in bytes). Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnTaskSetInitialPolicy()`	Purpose Defines the scheduler class and priority of the initial thread of a process. Parameters • [in] `task` – process handle. • [in] `policy` – scheduler class of the initial thread of the process. The parameter type is defined in the header file `sysroot-*-kos/include/thread/tidtype.h` from the KasperskyOS SDK. • [in] `priority` – priority of the initial thread of a process. The parameter type is defined in the header file `sysroot-*-kos/include/coresrv/thread/thread_api.h` from the KasperskyOS SDK. • [in,optional] `param` – pointer to the parameters of the initial thread scheduler class, or `RTL_NULL`. The data type for storing these parameters is defined in the header file `sysroot-*-kos/include/coresrv/thread/thread_api.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information In the `param` parameter, you can specify `RTL_NULL` if you need to define the default parameters for the scheduler class of the initial thread of the process. In the `param` parameter, you must specify `RTL_NULL` if the scheduler class of the initial thread of the process does not have any parameters.
`KnTaskReseedAslr()`	Purpose Defines the seed value for ASLR support in the defined process. Parameters • [in] `task` – process handle. • [in] `seed` – pointer to the buffer containing the seed value of the random number generator. • [in] `seedSize` – size (in bytes) of the buffer containing the seed value of the random number generator. Returned values

	If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnTaskGetElfSyms()`	Purpose Gets the addresses and sizes of the symbol table `.symtab` and string table `.strtab` for the calling process. Parameters • [out] `relocBase` – pointer to the ELF image load offset (in bytes). Add the ELF image load offset to the address of a symbol from the symbol table `.symtab` to get the address of this symbol in process memory. • [out] `syms` – pointer to the address of the symbol table `.symtab`. • [out] `symsCnt` – pointer to the size of the symbol table `.symtab` (in bytes). • [out] `strs` – pointer to the address of the string table `.strtab`. • [out] `strsCnt` – pointer to the size of the string table `.strtab` (in bytes). Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information If the symbol table `.symtab` and string table `.strtab` are not loaded into process memory by calling the `KnTaskLoadElfSyms()` function, the `KnTaskGetElfSyms()` function returns `rcOk` (if there are no other errors). In this case, the received addresses of the tables have `RTL_NULL` values, and the received sizes of the tables are zero. The function is intended for a mechanism that displays stack backtrace data and runs in a process and not in the kernel. This mechanism is currently under development.
`KnTaskGetIdByHandle()`	Purpose Gets the process ID (PID). Parameters • [in] `task` – process handle. • [out] `taskId` – pointer to the process ID. The ID type is defined in the header file `sysroot-*-kos/include/task/pidtype.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnTaskPanic()`	Purpose Initiates an exception that cannot be handled and leads to the process terminating. Parameters N/A Returned values N/A
`KnTaskTransferResource()`	Purpose Transfers a handle to a process that is not yet running. Parameters • [in] `task` – process handle. • [in] `srcHandle` – transferred handle. • [in] `srcBadge` – handle of the resource transfer context object.

	• [in] `dstRights` – permissions mask of the descendant of the transferred handle. • [out] `outDstHandle` – pointer to the descendant of the transferred handle (from the handle space of the process that received the handle). Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information In contrast to the `KosTaskTransferResource()` function from the API task.h, the `KnTaskTransferResource()` function does not copy the value of the descendant of the transferred handle to the SCP of the process to which the handle was transferred. The `KosTaskTransferResource()` function does this so that the process that received the descendant of the transferred handle can find it by calling the `KosTaskLookupResource()` function from the API task.h.

Initializing IPC transport for interprocess communication and managing IPC request processing (transport-kos.h, transport-kos-dispatch.h)

APIs are defined in the header files `transport-kos.h` and `transport-kos-dispatch.h` from the KasperskyOS SDK that are located at the path `sysroot-*-kos/include/coresrv/nk`.

API capabilities:

- Initialize IPC transport on the client side and on the server side.

  IPC transport is an add-on that works on top of system calls for sending and receiving IPC messages and works separately with the constant part and arena of IPC messages. Transport code works on top of this add-on.

- Start the loop for processing IPC requests on the server side.

- Copy data to the IPC message arena.

Information about API functions is provided in the tables below.

This section contains API usage examples. In these examples, programs acting as servers have the following formal specification:

FsDriver.edl

```
task class FsDriver
components {
    operationsComp : Operations
}
```

Operations.cdl

```
component Operations
endpoints {
    fileOperations : FileIface
}
```

FileIface.idl

```
package FileIface
interface {
    Open(in array<UInt8, 1024> path);
```

```
    Read(out sequence<UInt8, 2048> content);
}
```

Initializing IPC transport for interprocess communication

To initialize IPC transport for interaction with other processes, call the `NkKosTransport_Init()` or `NkKosTransportSync_Init()` function declared in the header file `transport-kos.h`.

Example use of the `NkKosTransport_Init()` function on the client side:

```
int main(int argc, const char *argv[])
{
    /* Declare the structure containing the IPC transport parameters */
    NkKosTransport driver_transport;
    /* Declare the proxy object. (The type of proxy object is automatically
     * generated transport code.) */
    struct FileIface_proxy file_operations_proxy;
    /* Declare the structures for saving the constant part of an IPC request and
     * IPC response for the endpoint method. (The types of structures are automatically
     * generated transport code.) */
    struct FileIface_Open_req req;
    struct FileIface_Open_res res;
    /* Get the client IPC handle and RIID */
    Handle driver_handle;
    rtl_uint32_t file_operations_riid;
    if (KnCmConnect("FsDriver", "operationsComp.fileOperations", INFINITE_TIMEOUT,
              &driver_handle, &file_operations_riid) == rcOk) {
        /* Initialize the structure containing the IPC transport parameters */
        NkKosTransport_Init(&driver_transport, driver_handle, NK_NULL, 0);
        /* Initialize the proxy object. (The proxy object initialization method is
         * is automatically generated transport code.) */
        FileIface_proxy_init(&file_operations_proxy, &driver_transport.base,
                             (nk_iid_t) file_operations_riid);
    }
...
    /* Call the endpoint method. (The method is automatically
     * generated transport code.) */
    strncpy(req.path, "/example/file/path", sizeof(req.path));
    if (FileIface_Open(file_operations_proxy.base, &req, NULL,
                       &res, NULL) != NK_EOK) {
...
    }
...
}
```

If a client needs to use several endpoints, the same number of proxy objects must be initialized. When initializing each proxy object, you need to specify the IPC transport that is associated through the client IPC handle with the relevant server. When initializing multiple proxy objects pertaining to the endpoints of one server, you can specify the same IPC transport that is associated with this server.

Example use of the `NkKosTransport_Init()` function on the server side:

```
int main(int argc, const char *argv[])
{
...
    /* Declare the structure containing the IPC transport parameters */
    NkKosTransport transport;
    /* Get the listener handle. (RIID
     * FsDriver_operationsComp_fileOperations_iid is
     * automatically generated transport code.) */
    Handle handle;
    char client[32];
    char endpoint[32];
    Retcode rc = KnCmListen(RTL_NULL, INFINITE_TIMEOUT, client, endpoint);
    if (rc == rcOk)
        rc = KnCmAccept(client, endpoint,
                        FsDriver_operationsComp_fileOperations_iid,
                        INVALID_HANDLE, &handle);
...
```

```
    /* Initialize the structure containing the IPC transport parameters */
    NkKosTransport_Init(&transport, handle, NK_NULL, 0);
...
    /* IPC request processing loop */
    do
    {
...
        /* Get the IPC request */
        rc = nk_transport_recv(&transport.base, ...);
        if (rc == NK_EOK) {
            /* Process the IPC request by calling the dispatcher. (The dispatcher
             * is automatically generated transport
             * code.) */
            rc = FsDriver_entity_dispatch(...);
            if (rc == NK_EOK) {
                /* Send an IPC response */
                rc = nk_transport_reply(&transport.base, ...);
            }
        }
    }
    while (rc == NK_EOK)
    return EXIT_SUCCESS;
}
```

If a server processes IPC requests received through multiple IPC channels, the following special considerations should be taken into account:

- If a listener handle is associated with all IPC channels, IPC interaction with all clients can use the same IPC transport associated with this listener handle.
- If IPC channels are associated with different listener handles, IPC interaction with each group of clients corresponding to the same listener handle must use a separate IPC transport associated with this listener handle. In this case, IPC requests can be processed in parallel threads if you are using a thread-safe implementation of endpoint methods.

The `NkKosTransportSync_Init()` function initializes IPC transport with support for interrupting the `Call()` and `Recv()` locking system calls. (For example, an interrupt of these calls may be required for correct termination of the process that is executing them.) To interrupt the `Call()` and `Recv()` system calls, use the API ipc_api.h.

The `NkKosSetTransportTimeouts()` function declared in the header file `transport-kos.h` defines the maximum lockout duration for `Call()` and `Recv()` system calls used for IPC transport.

Starting the IPC request processing loop

The IPC request processing loop on a server includes the following steps:

1. Receive an IPC request.
2. Process the IPC request.
3. Send an IPC response.

Each step of this loop can be completed separately by sequentially calling the `nk_transport_recv()`, dispatcher, and `nk_transport_reply()` functions. (The `nk_transport_recv()` and `nk_transport_reply()` functions are declared in the header file `sysroot-*-kos/include/nk/transport.h` from the KasperskyOS SDK.) You can also call the `NkKosTransport_Dispatch()`, `NkKosDoDispatch()` or `NkKosDoDispatchEx()` function in which this loop is completed in its entirety. (The `NkKosTransport_Dispatch()` function is declared in the header file `transport-kos.h`, and the `NkKosDoDispatch()` and `NkKosDoDispatchEx()` functions are declared in the header file `transport-kos-dispatch.h`.) Use of the `NkKosDoDispatch()` or `NkKosDoDispatchEx()` function is more convenient than the `NkKosTransport_Dispatch()` function because they require fewer preparatory operations (for example, you do not need to initialize IPC transport).

You can initialize the structure passed to the `NkKosDoDispatch()` or `NkKosDoDispatchEx()` function via the `info` parameter by using the macros defined in the header file `transport-kos-dispatch.h`. You can use the `syncHandle` parameter of the `NkKosDoDispatchEx()` function to interrupt the system call `Recv()`. You can use the `cb` parameter of the `NkKosDoDispatchEx()` function to define the callback functions that are called after completing the `Recv()` and `Reply()` system calls, and after exiting the IPC request processing loop (you do not need to define all functions). The callback function corresponding to the `enterDispatchProcessing` pointer is called immediately after the `Recv()` system call is completed. The callback function corresponding to the `leaveDispatchProcessing` pointer is called immediately after the `Reply()` system call is completed. The callback function corresponding to the `stopDispatchLoop` pointer is called immediately after exiting the IPC request processing loop. The `callbackContext` pointer is passed to each of these callback functions. These callback functions can be used to implement a thread pool, for example.

The `NkKosTransport_Dispatch()`, `NkKosDoDispatch()`, and `NkKosDoDispatchEx()` functions can be called from parallel threads if you are using a thread-safe implementation of endpoint methods.

Example use of the `NkKosDoDispatch()` function:

```
/* This function implements the endpoint method. */
static nk_err_t Open_impl(...)
{
...
}

/* This function implements the endpoint method. */
static nk_err_t Read_impl(...)
{
...
}

/* This function initializes the pointers to functions implementing the endpoint methods.
* (These pointers are used by the dispatcher to call functions implementing the
* endpoint methods.The types of structures are automatically generated
* transport code.
static struct FileIface *CreateFileOperations()
{
    static const struct FileIface_ops ops = {
        .Open = Open_impl,
        .Read = Read_impl
    };
    static struct FileIface impl = {
        .ops = &ops
    };
    return &impl;
}

int main(int argc, const char *argv[])
{
...
    /* Declare the structure that is required for the
     * NkKosDoDispatch() function to use transport code. */
    NkKosDispatchInfo info;
    /* Declare the stubs. (The types of stubs are automatically generated
     * transport code. */
    struct Operations_component component;
    struct FsDriver_entity entity;
    /* Get the listener handle */
    Handle handle = ServiceLocatorRegister("driver_connection", NULL, 0, &iid);
    assert(handle != INVALID_HANDLE);
    /* Initialize the stubs. (Methods for initializing stubs are
     * automatically generated transport code.) */Function
     * CreateFileOperations() is implemented by the developer of the
     * KasperskyOS-based solution to initialize
     * pointers to functions implementing the endpoint methods.) */
    Operations_component_init(&component, CreateFileOperations());
    FsDriver_entity_init(&entity, &component);
    /* Initialize the structure that is required for the
     * NkKosDoDispatch() function to use transport code. */
    info = NK_TASK_DISPATCH_INFO_INITIALIZER(FsDriver, entity);
    /* Start the IPC request processing loop */
    NkKosDoDispatch(handle, info);
    return EXIT_SUCCESS;
}
```

Example use of the `NkKosTransport_Dispatch()` function:

```
/* This function implements the endpoint method. */
static nk_err_t Open_impl(...)
{
...
}

/* This function implements the endpoint method. */
static nk_err_t Read_impl(...)
{
...
}

/* This function initializes the pointers to functions implementing the endpoint methods.
* (These pointers are used by the dispatcher to call functions implementing the
* endpoint methods.The types of structures are automatically generated
* transport code.
static struct FileIface *CreateFileOperations()
{
    static const struct FileIface_ops ops = {
        .Open = Open_impl,
        .Read = Read_impl
    };
    static struct FileIface impl = {
        .ops = &ops
    };
    return &impl;
}

int main(int argc, const char *argv[])
{
...
    /* Declare the structure containing the IPC transport parameters */
    NkKosTransport transport;
    /* Declare the stubs. (The types of stubs are automatically generated
     * transport code. */
    struct Operations_component component;
    struct FsDriver_entity entity;
    /* Declare the unions of the constant part of IPC requests and
     * IPC responses. (Types of unions are automatically generated
     * transport code.) */
    union FsDriver_entity_req req;
    union FsDriver_entity_res res;
    /* Declare the array for the IPC response arena. (The array size is
     * automatically generated transport code.) */
    char res_buffer[FsDriver_entity_res_arena_size];
    /* Declare and initialize the arena descriptor of the IPC response.
     * (The type of handle and its initialization macro are defined in the header file
     * sysroot-*-kos/include/nk/arena.h from the KasperskyOS SDK.) */
    struct nk_arena res_arena = NK_ARENA_INITIALIZER(res_buffer,
                                                     res_buffer + sizeof(res_buffer));
    /* Get the listener handle */
    Handle handle = ServiceLocatorRegister("driver_connection", NULL, 0, &iid);
    assert(handle != INVALID_HANDLE);
    /* Initialize the structure containing the IPC transport parameters */
    NkKosTransport_Init(&transport, handle, NK_NULL, 0);
    /* Initialize the stubs. (Methods for initializing stubs are
     * automatically generated transport code.) */Function
     * CreateFileOperations() is implemented by the developer of the
     * KasperskyOS-based solution to initialize
     * pointers to functions implementing the endpoint methods.) */
    Operations_component_init(&component, CreateFileOperations());
    FsDriver_entity_init(&entity, &component);
    /* Start the IPC request processing loop. (The dispatcher FsDriver_entity_dispatch
     * is automatically generated transport code.) */
    NkKosTransport_Dispatch(&transport.base, FsDriver_entity_dispatch,
                            &entity, &req, sizeof(FsDriver_entity_req),
                            RTL_NULL, &res, &res_arena);
    return EXIT_SUCCESS;
}
```

Copying data to the IPC message arena

To copy a string to the IPC message arena, call the `NkKosCopyStringToArena()` function declared in the header file `transport-kos.h`. This function reserves a segment of the arena and copies a string to this segment.

Example use of the `NkKosCopyStringToArena()` function:

```
static nk_err_t Read_impl(struct FileIface *self,
                          const struct FileIface_Read_req *req,
                          const struct nk_arena* req_arena,
                          struct FileIface_Read_res* res,
                          struct nk_arena* res_arena)
{
    /* Copy the string to the IPC response arena */
    if (NkKosCopyStringToArena(&res_arena, &res.content,
                               "CONTENT OF THE FILE") != rcOk) {
...
    }
    return NK_EOK;
}
```

Information about API functions

transport-kos.h functions

Function	Information about the function
`NkKosTransport_Init()`	Purpose Initializes IPC transport. Parameters • [out] `transport` – pointer to the structure containing the IPC transport parameters. • [in] `handle` – client or server IPC handle. • [in] `view` – parameter that must have the value `NK_NULL`. • [in] `size` – parameter that must have the value `0`. Returned values N/A
`NkKosTransportSync_Init()`	Purpose Initializes IPC transport with support for interrupting the `Call()` and/or `Recv()` system calls. Parameters • [out] `transport` – pointer to the structure containing the IPC transport parameters. • [in] `handle` – client or server IPC handle. • [in,optional] `callSyncHandle` – handle of the IPC synchronization object for `Call()` system calls, or `INVALID_HANDLE` if an interrupt of `Call()` system calls is not required. • [in,optional] `recvSyncHandle` – handle of the IPC synchronization object for `Recv()` system calls, or `INVALID_HANDLE` if an interrupt of `Recv()` system calls is not required. Returned values N/A
`NkKosSetTransportTimeouts()`	Purpose Defines the maximum lockout duration for `Call()` and `Recv()` system calls used for IPC transport. Parameters

	• [out] `transport` – pointer to the structure containing the IPC transport parameters. • [in] `recvTimeout` – maximum lockout duration for `Recv()` system calls in milliseconds, or `INFINITE_TIMEOUT` to define an unlimited lockout duration. • [in] `callTimeout` – maximum lockout duration for `Call()` system calls in milliseconds, or `INFINITE_TIMEOUT` to define an unlimited lockout duration. Returned values N/A
`NkKosTransport_Dispatch()`	Purpose Starts the IPC request processing loop. Parameters • [in] `transport` – pointer to the `base` field of the structure containing the IPC transport parameters. • [in] `dispatch` – pointer to the dispatcher (dispatch method) from the transport code. The dispatcher is named `<process class name>_entity_dispatch`. • [in] `impl` – pointer to the stub, which consists of a structure with the type `<process class name>_entity` from the transport code. The dispatcher uses this structure to get the pointers to functions implementing endpoint methods. • [out] `req` – pointer to the union with the type `<process class name>_entity_req` from the transport code. This union is intended for storing the constant part of IPC requests for any methods of endpoints provided by the server. • [in] `req_size` – maximum size (in bytes) of the constant part of IPC requests. It is defined as `sizeof(<process class name>_entity_req)`, where `<process class name>_entity_req` is the type from the transport code. • [in,out,optional] `req_arena` – pointer to the IPC request arena descriptor, or `RTL_NULL` if an IPC request arena is not in use. The type of handle is defined in the header file `sysroot-*-kos/include/nk/arena.h` from the KasperskyOS SDK. • [out] `res` – pointer to the union with the type `<process class name>_entity_res` from the transport code. This union is intended for storing the constant part of IPC responses for any methods of endpoints provided by the server. • [in,out,optional] `res_arena` – pointer to the IPC response arena descriptor, or `RTL_NULL` if an IPC response arena is not in use. The type of handle is defined in the header file `sysroot-*-kos/include/nk/arena.h` from the KasperskyOS SDK. Returned values If unsuccessful, it returns an error code.
`NkKosCopyStringToArena()`	Purpose Reserves a segment of the arena and copies a string to this segment. Parameters • [in,out] `arena` – pointer to the arena descriptor. The type of handle is defined in the header file `sysroot-*-kos/include/nk/arena.h` from the KasperskyOS SDK. • [out] `field` – pointer to the arena chunk descriptor where the string is copied. The type of handle is defined in the header file `sysroot-*-kos/include/nk/types.h`. • [in] `src` – pointer to the string to be copied to the IPC message arena. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

transport-kos-dispatch.h functions

Function	Information about the function
	Purpose

`NkKosDoDispatch()`	Starts the IPC request processing loop. Parameters • [in] `h` – server IPC handle. • [in] `info` – pointer to the structure containing the data required by the function to use transport code (including the names of types, sizes of the constant part and IPC message arena). Returned values N/A
`NkKosDoDispatchEx()`	Purpose Starts the IPC request processing loop. Parameters • [in] `h` – server IPC handle. • [in] `info` – pointer to the structure containing the data required by the function to use transport code (including the names of types, sizes of the constant part and IPC message arena). • [in,optional] `cb` – pointer to the structure containing pointers to the callback functions that are called after completing the `Recv()` and `Reply()` system calls, and after exiting the IPC request processing loop. In addition to the pointers to callback functions, this structure contains the pointer that is passed to these callback functions when they are called. You can set this parameter to `RTL_NULL` if you do not need to define callback functions. • [in,optional] `syncHandle` – handle of the IPC synchronization object, or `INVALID_HANDLE` if an interrupt of the `Recv()` system call is not required. Returned values N/A

Initializing IPC transport for querying the security module (transport-kos-security.h)

The API is defined in the header file `sysroot-*-kos/include/coresrv/nk/transport-kos-security.h` from the KasperskyOS SDK.

The API initializes IPC transport for querying the Kaspersky Security Module via the security interface. Transport code works on top of IPC transport.

Information about API functions is provided in the table below.

This section contains an API usage example. In this example, the program that queries the security module has the following formal specification:

Verifier.edl

```
task class Verifier
security Approve
```

Approve.idl

```
package Approve
interface {
    Check(in UInt32 port);
}
```

Fragment of the policy description in the example:

```
security.psl

...
security src=Verifier, method=Check { assert (message.port > 80) }
...
```

Using the API

To initialize IPC transport for querying the security module, call the `NkKosSecurityTransport_Init()` function.

Example use of the `NkKosSecurityTransport_Init()` function:

```
int main(void)
{
    /* Declare the structure containing the IPC transport parameters for querying the
     * security module */
    NkKosSecurityTransport security_transport;
    /* Declare the proxy object. (The type of proxy object is automatically
     * generated transport code.) */
    struct Approve_proxy security_proxy;
    /* Declare the structures for saving the constant part of an IPC request and IPC response for the
     * security interface method. (The types of structures are automatically generated
     * transport code.) */
    struct Approve_Check_req security_req;
    struct Approve_Check_res security_res;
    /* Initialize the structure containing the IPC transport parameters for querying the
     * security module */
    if (NkKosSecurityTransport_Init(&security_transport, NK_NULL, 0) == NK_EOK) {
        /* Initialize the proxy object. (The proxy object initialization method and the
         * security interface ID Verifier_securityIid are
         * automatically generated transport code.) */
        Approve_proxy_init(&security_proxy, &security_transport.base, Verifier_securityIid);
    }
...
    /* Call the security interface method. (The method is automatically generated
     * transport code.The method does not pass any data through the security_res parameter.
     * This parameter should be specified only if required by the method implementation.) */
    security_req.port = 80;
    nk_err_t result = Approve_Check(&security_proxy.base, &security_req,
                      NULL, &security_res, NULL);
    if (result == NK_EOK)
        fprintf(stderr, "Granted");
    if (result == NK_EPERM)
        fprintf(stderr, "Denied");
    else
        fprintf(stderr, "Error");
    return EXIT_SUCCESS;
}
```

If a process needs to use several security interfaces, the same number of proxy objects must be initialized by specifying the same IPC transport and the unique IDs of the security interfaces.

Information about API functions

transport-kos-security.h functions

Function	Information about the function
`NkKosSecurityTransport_Init()`	Purpose Initializes IPC transport for querying the Kaspersky Security Module through the security interface. Parameters

	• [out] `transport` – pointer to the structure containing the IPC transport parameters for querying the security module. • [in] `view` – parameter that must have the value `NK_NULL`. • [in] `size` – parameter that must have the value `0`. Returned values If successful, the function returns `NK_EOK`, otherwise it returns an error code.

Generating random numbers (random_api.h)

The API is defined in the header file `sysroot-*-kos/include/kos/random/random_api.h` from the KasperskyOS SDK.

The API generates random numbers and includes functions that can be used to ensure high entropy (high level of unpredictability) of the seed value of the random number generator. The *start number of the random number generator* (seed) determines the sequence of the generated random numbers. In other words, if the same seed value is set, the generator creates identical sequences of random numbers. (The entropy of these numbers is fully determined by the entropy of the seed value, which means that these numbers are not entirely random, but pseudorandom.)

The random number generator of one process does not depend on the random number generators of other processes.

Information about API functions is provided in the table below.

Using the API

To generate a sequence of random byte values, call the `KosRandomGenerate()` or `KosRandomGenerateEx()` function.

Example use of the `KosRandomGenerate()` function:

```
size_t random_number;
if (KosRandomGenerate(sizeof random_number, &random_number) == rcOk) {
...
}
```

The `KosRandomGenerateEx()` function gets the quality level of the generated random values through the output parameter `quality`. The quality level can be high or low. High-quality random values are random values that are generated while fulfilling all of the following conditions:

1. When the seed value is changed, at least one entropy source is registered and data is successfully received from all registered entropy sources.

   To register the entropy source, call the `KosRandomRegisterSrc()` function. This function uses the `callback` parameter to receive the pointer to a callback function of the following type:

   ```
   typedef Retcode (*KosRandomSeedMethod)(void      *context,
                                          rtl_size_t size,
                                          void      *output);
   ```

Using the `context` parameters, this callback function receives data with the specified `size` of bytes from the entropy source and writes this data to the `output` buffer. If the function returns `rcOk`, it is assumed that data was successfully received from the entropy source. An entropy source can be a digitalized signal of a sensor or a hardware-based random number generator, for example.

To deregister an entropy source, call the `KosRandomUnregisterSrc()` function.

2. The random number generator is initialized if the quality level was low before changing the seed value while fulfilling condition 1.

   If the quality level is low, it cannot be changed to high without initializing the random number generator.

   To initialize a random number generator, call the `KosRandomInitSeed()` function. The entropy of data passed through the `seed` parameter must be guaranteed by the calling process.

3. The counter of random byte values that were generated after changing the seed value while fulfilling condition 1 does not exceed the system-defined limit.

4. The counter of time that has elapsed since changing the seed value while fulfilling condition 1 does not exceed the system-defined limit.

If at least one of these conditions is not fulfilled, the generated random values are deemed low-quality values.

When a process is started, the seed value is automatically defined by the system. Then the seed value is modified when doing the following:

- Generating a sequence of random values.

  Each successful call of the `KosRandomGenerateEx()` function with the `size` parameter greater than zero and each successful call of the `KosRandomGenerate()` function results in a change of the seed value of the random number generator, but not each seed change results in the receipt of data from registered entropy sources. The registered entropy sources are used only when condition 3 or 4 is not fulfilled. If at least one entropy source is registered, the time counter for the change in the seed value is reset. If data from at least one entropy source is successfully received, the counter of generated random byte values is also reset.

  The quality level may change from high to low.

- Initializing a random number generator.

  Each successful call of the `KosRandomInitSeed()` function changes the seed value by using the data that is passed through the `seed` parameter and received from registered entropy sources. If at least one registered entropy source is available, the counters for generated byte values and the time of a change in the seed value are reset. Otherwise, only the counter of generated random byte values is reset.

  The quality level may change from high to low, and vice versa.

- Registering an entropy source.

  Each successful call of the `KosRandomRegisterSrc()` function changes the seed value by using the data only from the registered entropy source. The counter of the generated random byte values is also reset.

  The quality level may not change.

A possible scenario for generating high-quality random values includes the following steps:

1. Register at least one source of entropy by calling the `KosRandomRegisterSrc()` function.

2. Generate a sequence of random values by calling the `KosRandomGenerateEx()` function.

3. Check the quality level of the random values.

   If the quality level is low, initialize the random number generator by calling the `KosRandomInitSeed()` function, and proceed to step 2.

If the quality level is high, proceed to step 4.

4. Use random values.

To get the quality level without generating random values, call the `KosRandomGenerateEx()` function with the values `0` and `RTL_NULL` in the `size` and `output` parameters, respectively.

Information about API functions

random_api.h functions

Function	Information about the function
`KosRandomInitSeed()`	Purpose Initializes the random number generator. Parameters • [in] `seed` – pointer to the byte array that is used to change the seed value. The array must have a size of 32 bytes. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosRandomGenerate()`	Purpose Generates a sequence of random byte values. Parameters • [in] `size` – size (in bytes) of the buffer used to store the sequence. • [out] `output` – pointer to the buffer used to store the sequence. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosRandomGenerateEx()`	Purpose Generates a sequence of random byte values. Parameters • [in] `size` – size (in bytes) of the buffer used to store the sequence. • [out] `output` – pointer to the buffer used to store the sequence. • [out] `quality` – pointer to the boolean value that is true if the generated random values have high quality, and false if the generated random values have low quality. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosRandomRegisterSrc()`	Purpose Registers an entropy source. Parameters • [in] `callback` – pointer to the function that receives data from the entropy source to change the seed value. • [in,optional] `context` – pointer to the parameters passed to the function defined through the `callback` parameter, or `RTL_NULL` if there are no parameters.

	• [in] `size` – size (in bytes) of the data that should be received from the entropy source when calling the function defined through the `callback` parameter. The size must be at least 32 bytes. • [out] `handle` – address of the pointer used to deregister the entropy source. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosRandomUnregisterSrc()`	Purpose Deregisters the source of entropy. Parameters • [in] `handle` – pointer that is received when registering the source of entropy. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Getting and changing time values (time_api.h)

The API is defined in the header file `sysroot-*-kos/include/coresrv/time/time_api.h` from the KasperskyOS SDK.

Main capabilities of the API:

- Get and modify the system time
- Get the monotonic time that has elapsed since the moment the KasperskyOS kernel was started
- Get the resolution of the sources of system time and monotonic time

Information about API functions is provided in the table below.

time_api.h functions

Function	Information about the function
`KnGetSystemTimeRes()`	Purpose Gets the resolution of the system time source. Parameters • [out] `res` – pointer to the structure containing the resolution of the system time source (in nanoseconds) in the `nsec` field. The type of structure is defined in the header file `sysroot-*-kos/include/rtl/rtc.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnSetSystemTime()`	Purpose Sets the system time. Parameters • [in] `time` – pointer to the structure containing the `sec` field, which indicates the number of seconds that have elapsed since January 1, 1970, and the `nsec` field, which indicates the number of nanoseconds that have elapsed since the time defined in the `sec` field. The type of structure is defined in the header file `sysroot-*-kos/include/rtl/rtc.h` from the KasperskyOS SDK. Returned values

	If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnGetSystemTime()`	Purpose Gets the system time. Parameters • [out] `time` – pointer to the structure containing the `sec` field, which indicates the number of seconds that have elapsed since January 1, 1970, and the `nsec` field, which indicates the number of nanoseconds that have elapsed since the time defined in the `sec` field. The type of structure is defined in the header file `sysroot-*-kos/include/rtl/rtc.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnGetUpTimeRes()`	Purpose Gets the resolution of the source of monotonic time that has elapsed since the KasperskyOS kernel was started. Parameters • [out] `res` – pointer to the structure containing the `nsec` field, which indicates the resolution of the source of monotonic time (in nanoseconds) that has elapsed since the KasperskyOS kernel was started. The type of structure is defined in the header file `sysroot-*-kos/include/rtl/rtc.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnGetUpTime()`	Purpose Gets the monotonic time that has elapsed since the moment the KasperskyOS kernel was started. Parameters • [out] `time` – pointer to the structure containing the `sec` field, which indicates the number of seconds that have elapsed since the KasperskyOS kernel started, and the `nsec` field, which indicates the number of nanoseconds that have elapsed since the time defined in the `sec` field. The type of structure is defined in the header file `sysroot-*-kos/include/rtl/rtc.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnGetRtcTime()`	Purpose Gets the system time. Parameters • [out] `rt` – pointer to the structure containing the following time data: year, month, day, hours, minutes, seconds, and milliseconds. The type of structure is defined in the header file `sysroot-*-kos/include/rtl/rtc.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnGetMSecSinceStart()`	Purpose Gets the monotonic time that has elapsed since the moment the KasperskyOS kernel was started. Parameters N/A Returned values Monotonic time (in milliseconds) that has elapsed since the KasperskyOS kernel was started.
`KnAdjSystemTime()`	Purpose Starts gradual adjustment of the system time.

	Parameters • [in] `adj` – pointer to the structure containing the amount of time by which the system time must be adjusted (*sec*10^9+nsec* nanoseconds), or `RTL_NULL` if you do not need to start an adjustment but instead only need information about a previously run adjustment (through the `prev` parameter). The type of structure is defined in the header file `sysroot-*-kos/include/rtl/rtc.h` from the KasperskyOS SDK. • [in] `slew` – rate of system time adjustment (microseconds per second). • [out] `prev` – pointer to the structure containing the amount of time correction that remained (or remains if `RTL_NULL` was indicated in the `adj` parameter) for the already running gradual adjustment to fully complete (*sec*10^9+nsec* nanoseconds). The type of structure is defined in the header file `sysroot-*-kos/include/rtl/rtc.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information If a new adjustment is started before a previously running adjustment is finished, the previously running adjustment is interrupted.

Using notifications (notice_api.h)

The API is defined in the `sysroot-*-kos/include/coresrv/handle/notice_api.h` header file from the KasperskyOS SDK.

The API can track events associated with (both system and user) resources, and inform other processes and threads about events involving user resources.

Information about API functions is provided in the table below.

Using the API

The notification mechanism uses an event mask. An *event mask* is a value whose bits are interpreted as events that should be tracked or that have already occurred. An event mask has a size of 32 bits and consists of a general part and a specialized part. The common part describes events that are not specific to any resources. The specialized part describes events that are specific to certain resources. Specialized part flags for system resources and common part flags are defined in the `sysroot-*-kos/include/handle/event_descr.h` header file from KasperskyOS SDK. (For example, the common part flag `EVENT_OBJECT_DESTROYED` signifies resource termination, and the specialized part flag `EVENT_TASK_COMPLETED` signifies process termination.) Specialized part flags for a user resource are defined by the resource provider with the help of the `OBJECT_EVENT_SPEC()` and `OBJECT_EVENT_USER()` macros, which are defined in the `sysroot-*-kos/include/handle/event_descr.h` header file from the KasperskyOS SDK. The resource provider must export the public header files describing the flags of the specialized part.

The standard scenario for receiving notifications about events associated with resources consists of the following steps:

1. Creating a *notification receiver* (KasperskyOS kernel object that collects notifications) by calling the `KnNoticeCreate()` function.

The notification receiver ID is the pointer to the `KosObject` containing the handle of the notification receiver (for details about `KosObjects`, see Using KosObjects (objects.h)). Destruction of this object will cause the destruction of the notification receiver. If you need to create a copy of the notification receiver ID (for example, for use in another thread), you must increment the counter of links to the `KosObject` by calling the `KosRefObject()` function to ensure that this object exists while the created copy of the ID exists. If the copy of the notification receiver ID is no longer needed, you must decrement the counter of links to the `KosObject` by calling the `KosPutObject()` function to make sure that this object is destroyed if there are no other links.

2. Adding "resource—event mask" entries to the notification receiver to configure it to get notifications about events associated with relevant resources.

   To add a "resource—event mask" entry to the notification receiver, you need to call the `KnNoticeSubscribeToObject()` function. (The `OCAP_HANDLE_GET_EVENT` flag should be set in the handle permissions mask of the resource stated in the `object` parameter.) Several "resource—event mask" entries can be added for one resource, and the entry identifiers do not need to be unique. Tracked events for each "resource—event mask" entry should be defined with an event mask that may match one or several events.

   "Resource—event mask" entries added to the notification receiver can be fully or partially removed to prevent the receiver from getting notifications that match these entries. To remove all "resource—event mask" entries from the receiver, you need to call the `KnNoticeDropAndWake()` function. To remove from the receiver "resource—event mask" entries that refer to the same resource, you need to call the `KnNoticeUnsubscribeFromObject()` function. To remove from the receiver a "resource—event mask" entry with a specific identifier, you need to call the `KnNoticeUnsubscribeFromEvent()` function.

   "Resource—event mask" entries can be added to the notification receiver or removed from it throughout the entire lifetime of this notification receiver.

3. Extracting notifications from the receiver by using the `KnNoticeGetEvent()` or `KnNoticeGetEventEx()` function.

   You can set the waiting time for notifications to appear in the receiver when calling the `KnNoticeGetEvent()` or `KnNoticeGetEventEx()` function. Threads that are locked while waiting for notifications to appear in the receiver will resume when notifications appear, even if these notifications match "resource—event mask" entries that were added to the receiver during the waiting period.

   Threads that are locked while waiting for notifications to appear in the receiver will resume their execution and will not be locked upon subsequent calls of the `KnNoticeGetEvent()` or `KnNoticeGetEventEx()` function if all "resource—event mask" entries are removed from this receiver by calling the `KnNoticeDropAndWake()` function. If you add at least one "resource—event mask" entry to the notification receiver after calling the `KnNoticeDropAndWake()` function, threads that get notifications from that receiver will be locked again on `KnNoticeGetEvent()` or `KnNoticeGetEventEx()` function calls for the defined waiting time as long as there are no notifications. If all "resource—event mask" entries are removed from the notification receiver by using the `KnNoticeUnsubscribeFromObject()` and/or `KnNoticeUnsubscribeFromEvent()` function, threads waiting for notifications to appear in the receiver will not resume until the timeout elapses, and they will be locked during the waiting period upon subsequent calls of the `KnNoticeGetEvent()` or `KnNoticeGetEventEx()` function.

4. Terminate operations with the notification receiver by calling the `KnNoticeRelease()` function.

   The `KnNoticeRelease()` function removes all "resource—event mask" entries from the notification receiver and makes it impossible to add new ones. Threads that are locked while waiting for notifications to appear in the receiver will resume their execution and will not be locked upon subsequent calls of the `KnNoticeGetEvent()` or `KnNoticeGetEventEx()` function. In addition, the `KnNoticeRelease()` function decrements the counter of links to the `KosObject` containing the handle of the notification receiver. If there are no other links to this object, the notification receiver will be destroyed. Otherwise, the notification receiver will exist until all other links are deleted.

To notify other processes and/or threads about events associated with the user resource, you need to call the `KnNoticeSetObjectEvent()` function. Calling the function results in notifications appearing in receivers configured to get events defined with the `evMask` parameter and associated with the user resource defined with the `object` parameter. You cannot set flags of the general part of an event mask in the `evMask` parameter, because only the kernel can signal about events that match the general part of an event mask. If the process calling the `KnNoticeSetObjectEvent()` function created the user resource handle stated in the `object` parameter, you can set flags defined by the `OBJECT_EVENT_SPEC()` and `OBJECT_EVENT_USER()` macros in the `evMask` parameter. If the process calling the `KnNoticeSetObjectEvent()` function received the user resource handle stated in the `object` parameter from another process, you can set only those flags defined by the `OBJECT_EVENT_USER()` macro in the `evMask` parameter, while the permissions mask of the resulting handle must have a `OCAP_HANDLE_SET_EVENT` flag set.

Information about API functions

notice_api.h functions

Function	Information about the function
`KnNoticeCreate()`	Purpose Creates a notification receiver. Parameters • [out] `notice` – pointer to the identifier of the notification receiver. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnNoticeSubscribeToObject()`	Purpose Adds a "resource–event mask" entry to the notification receiver so that it can receive notifications about events associated with the defined resource and match the defined event mask. Parameters • [in] `notice` – identifier of the notification receiver. • [in] `object` – resource handle. • [in] `evMask` – event mask. • [in] `evId` – ID of the "resource–event mask" entry. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnNoticeGetEvent()`	Purpose Extracts notifications from the receiver. Parameters • [in] `notice` – identifier of the notification receiver. • [in] `msec` – time-out before notifications appearing in the receiver, in milliseconds, or `INFINITE_TIMEOUT` to set an unlimited time-out. • [in] `countMax` – maximum number of notifications extracted with one function call. • [out] `events` – pointer to a set of notifications that represent structures containing the identifier of a "resource—event mask" entry and the mask of the events associated with the resource. • [out] `count` – number of notifications extracted. Returned values

	If successful, the function returns `rcOk`, otherwise it returns an error code. If the time-out for notifications to appear in the receiver has elapsed, returns `rcTimeout`. If all "resource—event mask" entries are removed from the notification receiver by calling the `KnNoticeRelease()` or `KnNoticeDropAndWake()` function, the function returns `rcResourceNotFound`.
`KnNoticeGetEventEx()`	Purpose Extracts notifications from the receiver. Parameters • [in] `notice` – identifier of the notification receiver. • [in] `msec` – time-out before notifications appearing in the receiver, in milliseconds, or `INFINITE_TIMEOUT` to set an unlimited time-out. • [in] `countMax` – maximum number of notifications extracted with one function call. • [out] `events` – pointer to a set of notifications that represent structures containing the identifier of a "resource—event mask" entry and the counter of events associated with the resource. • [out] `count` – number of notifications extracted. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If the time-out for notifications to appear in the receiver has elapsed, returns `rcTimeout`. If all "resource—event mask" entries are removed from the notification receiver by calling the `KnNoticeRelease()` or `KnNoticeDropAndWake()` function, the function returns `rcResourceNotFound`.
`KnNoticeUnsubscribeFromObject()`	Purpose Removes from the notification receiver "resource—event mask" entries that match the specified resource to prevent the receiver from getting notifications about events that match these entries. Parameters • [in] `notice` – identifier of the notification receiver. • [in] `object` – resource handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information Notifications that correspond to the removed "resource—event mask" entries will be removed from the receiver.
`KnNoticeUnsubscribeFromEvent()`	Purpose Removes from the notification receiver "resource—event mask" entries with the specified identifier to prevent the receiver from getting notifications about events that match these entries. Parameters • [in] `notice` – identifier of the notification receiver. • [in] `evId` – ID of the "resource–event mask" entry. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information

	Notifications that correspond to the removed "resource—event mask" entries will be removed from the receiver.
`KnNoticeDropAndWake()`	Purpose Removes all "resource—event mask" entries from the specified notification receiver and resumes all threads that are waiting for notifications to appear in the specified receiver. Parameters • [in] `notice` – identifier of the notification receiver. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnNoticeRelease()`	Purpose Removes all "resource—event mask" entries from the specified notification receiver and makes it impossible to add new ones, resumes all threads waiting for notifications to appear in the specified receiver, and decrements the counter of links to the `KosObject` containing the handle of the specified notification receiver. Parameters • [in] `notice` – identifier of the notification receiver. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnNoticeSetObjectEvent()`	Purpose Signals the occurrence of events that are related to the defined user resource and match the defined event mask. Parameters • [in] `object` – user resource handle. • [in] `evMask` – mask of events to be signaled. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Dynamically creating IPC channels (cm_api.h, ns_api.h)

APIs are defined in the following header files from the KasperskyOS SDK:

- `sysroot-*-kos/include/coresrv/cm/cm_api.h`
- `sysroot-*-kos/include/coresrv/ns/ns_api.h`

The APIs dynamically create IPC channels.

Information about API functions is provided in the tables below.

Using the API

To ensure that servers can notify clients about the endpoints that they provide, the solution should include a name server, which is provided by the `NameServer` system program (executable file `sysroot-*-kos/bin/ns` from the KasperskyOS SDK). IPC channels from clients and servers to a name server can be statically created (these IPC channels must be named `kl.core.NameServer`). If this is not done, attempts will be made to dynamically create these IPC channels whenever clients and servers call the `NsCreate()` function. A name server does not have to be included in a solution if the clients initially already have information about the names of servers and the endpoints provided by these servers.

The names of endpoints and interfaces should be defined according to the formal specifications of solution components. (For information about the qualified name of an endpoint, see Binding methods of security models to security events.) Instead of the qualified name of an endpoint, you can use any conditional name of this endpoint. The names of clients and servers are defined in the init description. You can also get a process name by calling the `KnTaskGetName()` function from the API task_api.h.

Dynamic creation of an IPC channel on the server side includes the following steps:

1. Connect to the name server by calling the `NsCreate()` function.

2. Publish the provided endpoints on the name server by using the `NsPublishService()` function.

   To unpublish an endpoint, call the `NsUnPublishService()` function.

3. Receive a client request to create an IPC channel by calling the `KnCmListen()` function.

   The `KnCmListen()` function gets the first request in the queue without deleting this request but instead putting it at the end of the queue. If there is only one request in the queue, multiple consecutive calls of the `KnCmListen()` function will provide the same result. A request is deleted from the queue when the `KnCmAccept()` or `KnCmDrop()` function is called.

4. You can accept a client request to create an IPC channel by calling the `KnCmAccept()` function.

   To decline a client request, call the `KnCmDrop()` function.

   A listener handle is created when the `KnCmAccept()` function is called with the `INVALID_HANDLE` value in the `listener` parameter. If a listener handle is specified, the created server IPC handle will provide the capability to receive IPC requests over all IPC channels associated with this listener handle. (The first IPC channel associated with the listener handle is created when the `KnCmAccept()` function is called with the `INVALID_HANDLE` value in the `listener` parameter. The second and subsequent IPC channels associated with the listener handle are created during the second and subsequent calls of the `KnCmAccept()` function specifying the handle that was obtained during the first call.) In the `listener` parameter of the `KnCmAccept()` function, you can specify the listener handle received using the `KnHandleConnect()`, `KnHandleConnectEx()` and `KnHandleCreateListener()` functions from the API handle_api.h, and the `ServiceLocatorRegister()` function declared in the header file `sysroot-*-kos/include/coresrv/sl/sl_api.h` from the KasperskyOS SDK. In the `rsid` parameter of the `KnCmAccept()` function, you must specify the endpoint ID (RIID), which is a constant in the automatically generated transport code (for example: `FsDriver_operationsComp_fileOperations_iid`).

Dynamic creation of an IPC channel on the client side includes the following steps:

1. Connect to the name server by calling the `NsCreate()` function.

2. Find the server providing the required endpoint by using the `NsEnumServices()` function.

   To get a full list of endpoints with a defined interface, call the function several times while incrementing the index until you receive the `rcResourceNotFound` error.

3. Fulfill the request to create an IPC channel with the necessary server by calling the `KnCmConnect()` function.

You can connect multiple clients and servers to the name server. Each client and server can create multiple connections to the name server. A server can unpublish an endpoint that was published by another server.

To delete a connection to the name server, call the `NsDestroy()` function.

Deleting dynamically created IPC channels

A dynamically created IPC channel will be deleted when its client IPC handle and server IPC handle are closed.

Information about API functions

ns_api.h functions

Function	Information about the function
`NsCreate()`	Purpose Creates a connection to a name server. Parameters • [in,optional] `name` – pointer to the name of the name server process, or `RTL_NULL` to assign the default name (corresponds to the `NS_SERVER_NAME` macro value). • [in] `msecs` – timeout (in milliseconds) for creating a connection to the name server, or `INFINITE_TIMEOUT` to define an unlimited timeout. • [out] `ns` – pointer to the ID of the connection to the name server. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`NsDestroy()`	Purpose Deletes a connection to the name server. Parameters • [in] `ns` – ID of the connection to the name server. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`NsPublishService()`	Purpose Publishes an endpoint on a name server. Parameters • [in] `ns` – ID of the connection to the name server. • [in] `type` – pointer to the name of the endpoint interface. • [in,optional] `server` – pointer to the name of the server providing the endpoint, or `RTL_NULL` to use the name of the calling process. • [in] `service` – pointer to the qualified name of the endpoint. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`NsUnPublishService()`	Purpose Unpublishes an endpoint on a name server. Parameters • [in] `ns` – ID of the connection to the name server.

	• [in] `type` – pointer to the name of the endpoint interface. • [in,optional] `server` – pointer to the name of the server providing the endpoint, or `RTL_NULL` to use the name of the calling process. • [in] `service` – pointer to the qualified name of the endpoint. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`NsEnumServices()`	Purpose Enumerates the endpoints published on a name server. Parameters • [in] `ns` – ID of the connection to the name server. • [in] `type` – pointer to the name of the interface of endpoints. • [in] `index` – index for enumerating endpoints. Enumeration starts with zero. • [out] `server` – pointer to the buffer for the name of the server providing the endpoint. • [in] `serverSize` – buffer size (in bytes) for the name of the server providing the endpoint. • [out] `service` – pointer to the buffer for the qualified name of the endpoint. • [in] `serviceSize` – buffer size (in bytes) for the qualified name of the endpoint. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

cm_api.h functions

Function	Information about the function
`KnCmConnect()`	Purpose Requests to create an IPC channel with a server for use of the defined endpoint. Parameters • [in] `server` – pointer to the server name. • [in] `service` – pointer to the qualified name of the endpoint. • [in] `msecs` – timeout (in milliseconds) for fulfilling a request, or `INFINITE_TIMEOUT` to define an unlimited timeout. • [out] `handle` – pointer to the client IPC handle. • [out] `rsid` – pointer to the Runtime Implementation Identifier (RIID). Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnCmListen()`	Purpose Receives a client request to create an IPC channel for use of an endpoint. Parameters • [in] `filter` – fictitious parameter that must have the value `RTL_NULL`. • [in] `msecs` – timeout (in milliseconds) for the appearance of a client request, or `INFINITE_TIMEOUT` to define an unlimited timeout. • [out] `client` – pointer to the client name. • [out] `service` – pointer to the qualified name of the endpoint.

	Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnCmDrop()`	Purpose Rejects a client request to create an IPC channel for use of the defined endpoint. Parameters • [in] `client` – pointer to the client name. • [in] `service` – pointer to the qualified name of the endpoint. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnCmAccept()`	Purpose Accepts a client request to create an IPC channel for use of the defined endpoint. Parameters • [in] `client` – pointer to the client name. • [in] `service` – pointer to the qualified name of the endpoint. • [in] `rsid` – Runtime Implementation Identifier (RIID). • [in,optional] `listener` – listener handle, or `INVALID_HANDLE` if you need to create it. • [out] `handle` – pointer to the server IPC handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Using synchronization primitives (event.h, mutex.h, rwlock.h, semaphore.h, condvar.h)

The `libkos` library provides APIs that enable use of the following synchronization primitives:

- Events (`event.h`)
- Mutexes (`mutex.h`)
- Read-write locks (`rwlock.h`)
- Semaphores (`semaphore.h`)
- Conditional variables (`condvar.h`)

The header files are located in the KasperskyOS SDK at `sysroot-*-kos/include/kos`.

The APIs are intended for synchronizing threads that belong to the same process.

Events

An *event* is a synchronization primitive that is used to notify one or more threads about the fulfillment of a condition required by these threads. The notified thread waits for the event to switch from a non-signaling state to a signaling state, and the notifying thread changes the state of this event.

The standard API usage scenario for working with events includes the following steps:

1. An event is initialized via the `KosEventInit()` function call.

2. The event is used by threads:

   - The notified threads wait for the event to switch from non-signaling state to signaling state via the `KosEventWait()` or `KosEventWaitTimeout()` function call.

   - The notifying threads change the state of the event via the `KosEventSet()` and `KosEventReset()` function calls.

Information about the API event functions is provided in the table below.

event.h functions

Function	Information about the function
`KosEventInit()`	Purpose Initializes an event. The event is in a non-signaling state after it is initialized. Parameters • [out] `event` – pointer to the event. The event type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values N/A
`KosEventSet()`	Purpose Sets the event state to signaling. Parameters • [out] `event` – pointer to the event. The event type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values N/A
`KosEventReset()`	Purpose Sets the event state to non-signaling. Parameters • [out] `event` – pointer to the event. The event type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values N/A
`KosEventWait()`	Purpose Waits for the event to change its state from non-signaling to signaling. Parameters

	• [in,out] `event` – pointer to the event. The event type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. • [in] `reset` – value that defines whether the event state should be changed set to non-signaling after the time-out has elapsed (`rtl_true` – yes, `rtl_false` – no). The parameter type is defined in the `sysroot-*-kos/include/rtl/stdbool.h` header file from the KasperskyOS SDK. Returned values N/A
`KosEventWaitTimeout()`	Purpose Waits on the event to change its state from non-signaling to signaling for a period that does not exceed the specified time. Parameters • [in,out] `event` – pointer to the event. The event type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. • [in] `reset` – value that defines whether the event state should be changed set to non-signaling after the time-out has elapsed (`rtl_true` – yes, `rtl_false` – no). The parameter type is defined in the `sysroot-*-kos/include/rtl/stdbool.h` header file from the KasperskyOS SDK. • [in] `mdelay` – time-out (in milliseconds) or `INFINITE_TIMEOUT` to set an unlimited time-out. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Returns `rcTimeout` if the time-out has elapsed.

Mutexes

A *mutex* is a synchronization primitive that ensures mutually exclusive execution of *critical sections* (areas of code where resources shared between threads are queried). One thread captures the mutex and executes a critical section. Meanwhile, other threads wait for the mutex to be freed and attempt to capture this mutex to execute other critical sections. A mutex can be freed only by the specific thread that captured it. You can use a *recursive mutex*, which can be captured by the same thread multiple times.

The standard API usage scenario for working with mutexes includes the following steps:

1. A mutex is initialized via the `KosMutexInit()` or `KosMutexInitEx()` function call.
2. The mutex is used by threads:
   a. The mutex is captured via the `KosMutexTryLock()`, `KosMutexLock()` or `KosMutexLockTimeout()` function call.
   b. The mutex is freed via the `KosMutexUnlock()` function call.

Information about the API mutex functions is provided in the table below.

mutex.h functions

Function	Information about the function
`KosMutexInit()`	Purpose Initializes a non-recursive mutex. Parameters • [out] `mutex` – pointer to the mutex. The mutex type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK.

	Returned values N/A
`KosMutexInitEx()`	Purpose Initializes a mutex. Parameters • [out] `mutex` – pointer to the mutex. The mutex type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. • [in] `recursive` – value that defines whether the mutex should be recursive (`1` – yes, `0` – no). Returned values N/A
`KosMutexTryLock()`	Purpose Acquires the mutex. If the mutex is already acquired, returns control rather than waits for the mutex to be released. Parameters • [in,out] `mutex` – pointer to the mutex. The mutex type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If the mutex is already acquired, returns `rcBusy`.
`KosMutexLock()`	Purpose Acquires the mutex. If the mutex is already acquired, waits indefinitely for it to be released. Parameters • [in,out] `mutex` – pointer to the mutex. The mutex type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values N/A
`KosMutexUnlock()`	Purpose Releases the mutex. Parameters • [in,out] `mutex` – pointer to the mutex. The mutex type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values N/A
`KosMutexLockTimeout()`	Purpose Acquires the mutex. If the mutex is already acquired, waits for it to be released for a period that does not exceed the specified time. Parameters • [in,out] `mutex` – pointer to the mutex. The mutex type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK.

	• [in] `mdelay` – time-out (in milliseconds) or `INFINITE_TIMEOUT` to set an unlimited time-out. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Returns `rcTimeout` if the time-out has elapsed.

Read-write locks

A *read-write lock* is a synchronization primitive used to allow access to resources shared between threads: write access for one thread or read access for multiple threads at the same time.

The standard API usage scenario for working with read-write locks includes the following steps:

1. A read-write lock is initialized by the `KosRWLockInit()` function call.
2. The read-write lock is used by threads:
   a. The read-write lock is captured for write operations (via the `KosRWLockWrite()` or `KosRWLockTryWrite()` function call) or for read operations (via the `KosRWLockRead()` or `KosRWLockTryRead()` function call).
   b. The read-write lock is freed via the `KosRWLockUnlock()` function call.

Information about the API read-write lock functions is provided in the table below.

rwlock.h functions

Function	Information about the function
`KosRWLockInit()`	Purpose Initializes a read-write lock. Parameters • [out] `rwlock` – pointer to a read-write lock. The read-write lock type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values N/A
`KosRWLockRead()`	Purpose Acquires the read-write lock for reading. If the read-write lock is already acquired for writing, or if there are threads waiting on the lock to be acquired for writing, waits indefinitely for the lock to be released. Parameters • [in,out] `rwlock` – pointer to the read-write lock. The read-write lock type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values N/A
`KosRWLockTryRead()`	Purpose Acquires the read-write lock for reading. If the read-write lock is already acquired for writing, or if there are threads waiting on the lock to be acquired for writing, returns control, rather than waits for the lock to be released.

	Parameters • [in,out] `rwlock` – pointer to the read-write lock. The read-write lock type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosRWLockWrite()`	Purpose Acquires the read-write lock for writing. If the read-write lock is already acquired for writing or reading, waits indefinitely for the lock to be released. Parameters • [in,out] `rwlock` – pointer to the read-write lock. The read-write lock type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values N/A
`KosRWLockTryWrite()`	Purpose Acquires the read-write lock for writing. If the read-write lock is already acquired for writing or reading, returns control, rather than waits for the lock to be released. Parameters • [in,out] `rwlock` – pointer to the read-write lock. The read-write lock type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosRWLockUnlock()`	Purpose Releases the read-write lock. Parameters • [in,out] `rwlock` – pointer to the read-write lock. The read-write lock type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values N/A Additional information If the read-write lock is acquired for reading, it remains acquired for reading until released by every reading thread.

Semaphores

A *semaphore* is a synchronization primitive that is based on a counter whose value can be atomically modified. The value of the counter normally reflects the number of available resources shared between threads. To execute a critical section, the thread waits until the counter value becomes greater than zero. If the counter value is greater than zero, it is decremented by one and the thread executes the critical section. After the critical section is executed, the thread signals the semaphore and the counter value is increased.

The standard API usage scenario for working with semaphores includes the following steps:

1. A semaphore is initialized via the `KosSemaphoreInit()` function call.

2. The semaphore is used by threads:

   a. They wait for the semaphore via the `KosSemaphoreWait()`, `KosSemaphoreWaitTimeout()` or `KosSemaphoreTryWait()` function call.

   b. The semaphore is signaled via the `KosSemaphoreSignal()` or `KosSemaphoreSignalN()` function call.

3. Deallocating semaphore resources by calling the `KosSemaphoreDeinit()` function.

Information about the API semaphore functions is provided in the table below.

semaphore.h functions

Function	Information about the function
`KosSemaphoreInit()`	Purpose Initializes a semaphore. Parameters • [out] `semaphore` – pointer to the semaphore. The semaphore type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. • [in] `count` – counter value. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If the value of the `count` parameter exceeds the `KOS_SEMAPHORE_VALUE_MAX` constant, returns `rcInvalidArgument`. (The `KOS_SEMAPHORE_VALUE_MAX` constant is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK.)
`KosSemaphoreDeinit()`	Purpose Deallocates semaphore resources. Parameters • [in] `semaphore` – pointer to the semaphore. The semaphore type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If there are threads waiting on the semaphore, returns `rcBusy`.
`KosSemaphoreSignal()`	Purpose Signals the semaphore and increases the counter by one. Parameters • [in, out] `semaphore` – pointer to the semaphore. The semaphore type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosSemaphoreSignalN()`	Purpose Signals the semaphore and increases the counter by the specified number. Parameters • [in, out] `semaphore` – pointer to the semaphore. The semaphore type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK.

	• [in] `n` – natural number by which to increase the counter. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosSemaphoreWaitTimeout()`	Purpose Waits on the semaphore for a period that does not exceed the specified time. Parameters • [in, out] `semaphore` – pointer to the semaphore. The semaphore type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. • [in] `mdelay` – semaphore time-out in milliseconds or `INFINITE_TIMEOUT` to set an unlimited time-out. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Returns `rcTimeout` if the time-out has elapsed.
`KosSemaphoreWait()`	Purpose Waits on the semaphore indefinitely. Parameters • [in, out] `semaphore` – pointer to the semaphore. The semaphore type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosSemaphoreTryWait()`	Purpose Waits on the semaphore. If the semaphore counter has a zero value, returns control, rather than waits for the semaphore counter to increase. Parameters • [in, out] `semaphore` – pointer to the semaphore. The semaphore type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If the semaphore counter has a zero value, returns `rcBusy`.

Conditional variables

A *conditional variable* is a synchronization primitive that is used to notify one or more threads about the fulfillment of a condition required by these threads. A conditional variable is used together with a mutex. The notifying and notified threads capture a mutex to execute critical sections. During execution of a critical section, the notified thread verifies that its required condition was fulfilled (for example, the data has been prepared by the notifying thread). If the condition is fulfilled, the notified thread executes the critical section and frees the mutex. If the condition is not fulfilled, the notified thread is locked at the conditional variable and waits for the condition to be fulfilled. When this happens, the mutex is automatically freed. During execution of a critical section, the notifying thread verifies fulfillment of the condition required by the notified thread. If the condition is fulfilled, the notifying thread signals this fulfillment through the conditional variable and frees the mutex. The notified thread that was locked and waiting for the fulfillment of its required condition resumes execution of the critical section while automatically capturing the mutex. After the critical section is executed, the notified thread frees the mutex.

The standard API usage scenario for working with conditional variables includes the following steps:

1. The conditional variable and mutex are initialized.

   To initialize a conditional variable, you need to call the `KosCondvarInit()` function.

2. The conditional variable and mutex are used by threads.

Use of a conditional variable and mutex by notified threads includes the following steps:

1. The mutex is captured.

2. Condition fulfillment is verified.

3. The `KosCondvarWait()` or `KosCondvarWaitTimeout()` function is called to wait for condition fulfillment.

   After the `KosCondvarWait()` or `KosCondvarWaitTimeout()` function is returned, you normally need to re-verify that the condition is fulfilled because another notified thread also received the signal and may have voided this condition again. (For example, another thread could have extracted the data prepared by the notifying thread). To do so, you need to use the following construct:

   ```
   while(<condition>)
   <call of KosCondvarWait() or KosCondvarWaitTimeout()>
   ```

4. The mutex is freed.

Use of a conditional variable and mutex by notifying threads includes the following steps:

1. The mutex is captured.

2. Condition fulfillment is verified.

3. Fulfillment of the condition is signaled via the `KosCondvarSignal()` or `KosCondvarBroadcast()` function call.

4. The mutex is freed.

Information about the API conditional variable functions is provided in the table below.

condvar.h functions

Function	Information about the function
`KosCondvarInit()`	Purpose Initializes a conditional variable. Parameters • [out] `condvar` – pointer to the conditional variable. The conditional variable type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values N/A
`KosCondvarWaitTimeout()`	Purpose Waits for condition fulfillment for a period that does not exceed the specified time. Parameters

	• [in] `condvar` – pointer to the conditional variable. The conditional variable type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. • [in,out] `mutex` – pointer to the mutex. The mutex type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. • [in] `mdelay` – condition fulfillment time-out in milliseconds, or `INFINITE_TIMEOUT` to set an unlimited time-out. Returned values Returns `rcOk` if successful. Returns `rcTimeout` if the time-out has elapsed.
`KosCondvarWait()`	Purpose Waits indefinitely for condition fulfillment. Parameters • [in] `condvar` – pointer to the conditional variable. The conditional variable type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. • [in,out] `mutex` – pointer to the mutex. The mutex type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosCondvarSignal()`	Purpose Signals condition fulfillment to one of the threads waiting for it. Parameters • [in, out] `condvar` – pointer to the conditional variable. The conditional variable type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values N/A
`KosCondvarBroadcast()`	Purpose Signals condition fulfillment to all threads waiting for it. Parameters • [in, out] `condvar` – pointer to the conditional variable. The conditional variable type is defined in the `sysroot-*-kos/include/kos/sync_types.h` header file from the KasperskyOS SDK. Returned values N/A

Managing I/O memory isolation (iommu_api.h)

The API is defined in the `sysroot-*-kos/include/coresrv/iommu/iommu_api.h` header file from the KasperskyOS SDK.

The API is intended for managing the isolation of physical memory regions used by devices on a PCIe bus for DMA. (Isolation is provided by the IOMMU.)

Information about API functions is provided in the table below.

Using the API

A device on the PCIe bus cannot use DMA unless the device is attached to the IOMMU domain. After a device is attached to the IOMMU domain, the device can access all DMA buffers that are associated with this IOMMU domain. A device can be attached to only one IOMMU domain at a time, but multiple devices can be attached to the same IOMMU domain. A DMA buffer can be associated with multiple IOMMU domains at the same time. Each IOMMU domain is associated with only one process, but multiple IOMMU domains can be associated with the same process. The capability to create multiple IOMMU domains associated with the same process allows you to restrict access to DMA buffers for different devices managed by the same process.

The API lets you create IOMMU domains, attach devices on a PCIe bus to IOMMU domains, and detach devices on the PCIe bus from IOMMU domains. A device is normally attached to an IOMMU domain when its driver is initialized. A device is usually detached from an IOMMU domain when errors are encountered during driver initialization or driver finalization.

A DMA buffer is associated with an IOMMU domain when calling the `KnIoDmaBegin()` and `KnIoDmaBeginEx()` functions that are included in the dma.h API. The `KnIoDmaBegin()` function associates a DMA buffer with an automatically created IOMMU domain. Each process may be associated with only one such IOMMU domain. This IOMMU domain is created after the first successful call of the `KnIommuAttachDevice()` function and exists throughout the entire lifetime of the process. The `KnIoDmaBeginEx()` function associates a DMA buffer with the IOMMU domain that was created by calling the `KnIommuCreateDomain()` function. Each process may be associated with multiple IOMMU domains that were created by `KnIommuCreateDomain()` function calls. These IOMMU domains may be removed prior to process termination by closing their handles.

To attach a device to an automatically created IOMMU domain or detach it, you must call the `KnIommuAttachDevice()` or `KnIommuDetachDevice()` function, respectively.

To attach a device to an IOMMU domain that was created by calling the `KnIommuCreateDomain()` function or detach it, you must call the `KnIommuAttachDeviceToDomain()` or `KnIommuDetachDeviceFromDomain()` function, respectively.

Information about API functions

iommu_api.h functions

Function	Information about the function
`KnIommuAttachDevice()`	Purpose Attaches a device on a PCIe bus to the automatically created IOMMU domain associated with the calling process. Parameters • [in] `bdf` – address of the device on the PCIe bus in BDF format. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information If IOMMU is not enabled, `rcOk` is returned.
`KnIommuDetachDevice()`	Purpose Detaches a device on a PCIe bus from the automatically created IOMMU domain associated with the calling process. Parameters • [in] `bdf` – address of the device on the PCIe bus in BDF format.

	Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information If IOMMU is not enabled, `rcOk` is returned.
`KnIommuCreateDomain()`	Purpose Creates an IOMMU domain associated with the calling process. Parameters • [out] `domain` – pointer to the IOMMU domain handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnIommuAttachDeviceToDomain()`	Purpose Attaches a device on a PCIe bus to the defined IOMMU domain associated with the calling process. Parameters • [in] `bdf` – address of the device on the PCIe bus in BDF format. • [in] `domain` – IOMMU domain handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnIommuDetachDeviceFromDomain()`	Purpose Detaches a device on a PCIe bus from the defined IOMMU domain associated with the calling process. Parameters • [in] `bdf` – address of the device on the PCIe bus in BDF format. • [in] `domain` – IOMMU domain handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Using queues (queue.h)

The API is defined in the header file `sysroot-*-kos/include/kos/queue.h` from the KasperskyOS SDK.

The API sets up data exchange between threads owned by one process via a queuing mechanism that does not lock threads. In other words, you can add or extract elements of a queue without locking other threads that add or extract elements of this queue.

Information about API functions is provided in the table below.

Using the API

The standard scenario for API usage includes the following steps:

1. Create a queue abstraction.

A queue abstraction consists of a structure containing queue metadata and a queue buffer intended for storing elements of the queue. A queue buffer is logically divided into equal segments, each of which is intended for an individual element of the queue. The number of segments in a queue buffer matches the maximum number of elements in the queue. The alignment of segment addresses corresponds to the data types of elements in the queue.

To complete this step, call the `KosQueueCreate()` function. This function can allocate memory for the queue buffer or use already allocated memory. The size of the already allocated memory must be sufficient to accommodate the maximum number of elements in the queue. Also take into account that the size of a segment in the queue buffer is rounded to the next largest multiple of the alignment value defined through the `objAlign` parameter. The initial address of the already allocated memory must also be aligned to correspond to the data types of queue elements. If the memory address alignment specified in the `buffer` parameter is less than the value defined through the `objAlign` parameter, the function returns `RTL_NULL`.

2. Exchange data between threads by adding and extracting elements of the queue.

   To add one element to the end of the queue, you must reserve a segment in the queue buffer by calling the `KosQueueAlloc()` function, copy this element to the reserved segment, and call the `KosQueuePush()` function.

   To add a sequence of elements to the end of the queue, you must reserve the necessary number of segments in the queue buffer via `KosQueueAlloc()` function calls, copy the elements of this sequence to the reserved segments, and call the `KosQueuePushArray()` function. The order of elements in a sequence is not changed after this sequence is added to the queue. In other words, elements are added to the queue in the same order in which the pointers to reserved segments in the queue buffer are put into the array that is passed through the `objs` parameter of the `KosQueuePushArray()` function.

   To extract the first element of the queue, you must call the `KosQueuePop()` function. This function returns the pointer to the reserved segment in the queue buffer that contains the first element of the queue. After using an extracted element (for example, after checking or saving the value of an element), you must free the queue buffer segment occupied by this element. To do so, call the `KosQueueFree()` function.

   To clear the queue and free all registered segments in the queue buffer, you must call the `KosQueueFlush()` function.

3. Delete the queue abstraction.

   To complete this step, call the `KosQueueDestroy()` function. This function deletes the queue buffer if only the memory for this buffer was allocated by the `KosQueueCreate()` function. Otherwise, you must separately delete the queue buffer.

Information about API functions

queue.h functions

Function	Information about the function
`KosQueueCreate()`	Purpose Creates a queue abstraction. Parameters • [in] `objCount` – maximum number of elements in the queue. • [in] `objSize` – size (in bytes) of an element in the queue. • [in] `objAlign` – alignment of segment addresses in the queue buffer. The addresses of segments in the queue buffer may be unaligned (*objAlign=1*) or aligned (*objAlign=2,4,...,2^N*) to the boundary of a *2^N*-byte sequence (for example, two-byte or four-byte). • [in,optional] `buffer` – pointer to the allocated memory for the queue buffer, or `RTL_NULL` to automatically allocate the memory. Returned values

	If successful, the function returns the queue abstraction ID, otherwise it returns `RTL_NULL`.
`KosQueueDestroy()`	Purpose Deletes a queue abstraction. Parameters • [in] `queue` – queue abstraction ID. Returned values N/A
`KosQueueAlloc()`	Purpose Reserves a segment in the queue buffer. Parameters • [in] `queue` – queue abstraction ID. Returned values If successful, the function returns the pointer to the reserved segment in the queue buffer, otherwise it returns `RTL_NULL`.
`KosQueueFree()`	Purpose Resets the reservation of the defined segment in the queue buffer. Parameters • [in] `queue` – queue abstraction ID. • [in] `obj` – pointer to the reserved segment in the queue buffer. Returned values N/A
`KosQueuePush()`	Purpose Adds an element to the end of the queue. Parameters • [in] `queue` – queue abstraction ID. • [in] `obj` – pointer to the reserved segment in the queue buffer. Returned values N/A
`KosQueuePushArray()`	Purpose Adds a sequence of elements to the end of the queue. Parameters • [in] `queue` – queue abstraction ID. • [in] `objs` – array of pointers to reserved segments in the queue buffer. • [in] `count` – number of elements in the sequence. Returned values N/A
`KosQueuePop()`	Purpose Extracts the first element of the queue.

	Parameters • [in] `queue` – queue abstraction ID. • [in] `timeout` – timeout (in milliseconds) for an element to appear in the queue, or `INFINITE_TIMEOUT` to set an unlimited timeout. Returned values If successful, this function returns the pointer to the reserved segment in the queue buffer containing the first element of the queue. Otherwise it returns `RTL_NULL`.
`KosQueueFlush()`	Purpose Clears the queue and resets the reservation of all registered segments in the queue buffer. Parameters • [in] `queue` – queue abstraction ID. Returned values N/A

Using memory barriers (barriers.h)

This API is defined in the header file `sysroot-*-kos/include/coresrv/io/barriers.h` from the KasperskyOS SDK.

The API sets barriers for reading from memory and/or writing to memory. A *memory barrier* is an instruction for a compiler or processor that guarantees that memory access operations specified in source code before setting a barrier will be executed before the memory access operations specified in source code after setting a barrier. Use of memory barriers is required if the specific order of memory write and memory read operations is important. Otherwise, the optimization mechanisms of a compiler and/or processor could cause these operations to be executed in a different order than the order specified in the source code.

Information about API functions is provided in the table below.

barriers.h functions

Function	Information about the function
`IoReadBarrier()`	Purpose Sets a barrier for reading from memory. Parameters N/A Returned values N/A
`IoWriteBarrier()`	Purpose Sets a barrier for writing to memory. Parameters N/A Returned values N/A
`IoReadWriteBarrier()`	Purpose

	Sets a barrier for writing to memory and reading from memory. Parameters N/A Returned values N/A

Executing system calls (syscalls.h)

This API is defined in the header file `sysroot-*-kos/include/coresrv/syscalls.h` from the KasperskyOS SDK.

The API allows execution of the `Call()`, `Recv()`, and `Reply()` system calls for sending and receiving IPC messages.

Information about API functions is provided in the table below.

Using the API

Pointers to buffers containing the constant part and arena of IPC messages are passed to API functions by using an IPC message header whose type is defined in the header file `sysroot-*-kos/include/ipc/if_rend.h` from the KasperskyOS SDK. Prior to API function calls, the headers of IPC messages must be bound to buffers containing the constant part and arena of IPC messages. To do so, use the `PackInMsg()` and `PackOutMsg()` functions that are declared in the header file `sysroot-*-kos/include/services/rtl/nk_msg.h` from the KasperskyOS SDK.

The `Call()`, `CallEx()`, `Recv()`, and `RecvEx()` functions lock execution of the calling thread while waiting for the system calls to complete. The `CallEx()` and `RecvEx()` functions let you define the timeout for completion of a system call. When this timeout is reached, an uncompleted system call is interrupted and the thread that is waiting for its completion resumes execution. A system call is also interrupted if an error occurs during its execution (such as an error due to termination of a server process). If a thread waiting on the completion of a system call is terminated externally, this system call is also interrupted. A system call executed by the `CallEx()` or `RecvEx()` function can be interrupted (for example, for correct termination of a process) by using the API ipc_api.h.

If a system call was interrupted using the API ipc_api.h, the `CallEx()` and `RecvEx()` functions return the error code `rcIpcInterrupt`. If IPC message transmission is prohibited by security mechanisms (such as the Kaspersky Security Module or a capability-based security mechanism implemented by the KasperskyOS kernel), the `Call()`, `CallEx()`, and `Reply()` functions return the error code `rcSecurityDisallow`.

Information about API functions

syscalls.h functions

Function	Information about the function
`Call()`	Purpose Executes the `Call()` system call with an unlimited timeout for its completion. Parameters • [in] `handle` – client IPC handle.

	• [in] `msgOut` – pointer to the header of IPC requests. • [in,out] `msgIn` – pointer to the header of IPC responses. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`CallEx()`	Purpose Executes the `Call()` system call with a defined timeout for its completion and the capability to interrupt its execution. Parameters • [in] `handle` – client IPC handle. • [in] `msgOut` – pointer to the header of IPC requests. • [in,out] `msgIn` – pointer to the header of IPC responses. • [in] `mdelay` – timeout (in milliseconds) for completion of a `Call()` system call, or `INFINITE_TIMEOUT` to define an unlimited timeout. • [in,optional] `syncHandle` – handle of the IPC synchronization object, or `INVALID_HANDLE` if an interrupt of the `Call()` system call is not required. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`Reply()`	Purpose Executes the `Reply()` system call. Parameters • [in] `handle` – server IPC handle. • [in] `msgOut` – pointer to the header of IPC responses. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`Recv()`	Purpose Executes the `Recv()` system call with an unlimited timeout for its completion. Parameters • [in] `handle` – server IPC handle. • [in,out] `msgIn` – pointer to the header of IPC requests. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`RecvEx()`	Purpose Executes the `Recv()` system call with a defined timeout for its completion and the capability to interrupt its execution. Parameters • [in] `handle` – server IPC handle. • [in,out] `msgIn` – pointer to the header of IPC responses. • [in] `mdelay` – timeout (in milliseconds) for completion of a `Recv()` system call, or `INFINITE_TIMEOUT` to define an unlimited timeout. • [in,optional] `syncHandle` – handle of the IPC synchronization object, or `INVALID_HANDLE` if an interrupt of the `Recv()` system call is not required.

	Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

IPC interrupt (ipc_api.h)

This API is defined in the header file `sysroot-*-kos/include/coresrv/ipc/ipc_api.h` from the KasperskyOS SDK.

The API interrupts the `Call()` and `Recv()` system calls if one or more process threads are locked while waiting for these system calls to complete. For example, you may need to interrupt these system calls to correctly terminate a process so that threads waiting for the completion of these system calls can resume execution.

Information about API functions is provided in the table below.

Using the API

The API interrupts system calls in process threads that were locked after the `CallEx()` or `RecvEx()` function was called from the API syscalls.h if these functions were called while specifying the IPC synchronization object handle in the `syncHandle` parameter. To create an IPC synchronization object, call the `KnIpcCreateSyncObject()` function. (The handle of an IPC synchronization object cannot be transferred to another process because the necessary flag for this operation is not set in the permissions mask of this handle.)

The `KnIpcSetInterrupt()` function switches an IPC synchronization object to a state that allows interruption of the system calls in those process threads that have been locked after a `CallEx()` or `RecvEx()` function call specifying the handle of this IPC synchronization object in the `syncHandle` parameter. A system call can be interrupted only during certain stages of its execution. A system call that is executed by the `CallEx()` function can be interrupted only when the server has not yet received a `Recv()` or `RecvEx()` function call for the IPC channel whose client IPC handle was specified during the `CallEx()` function call. A system call executed by the `RecvEx()` function can be interrupted only while waiting for an IPC request from a client.

The `KnIpcClearInterrupt()` function cancels the action of the `KnIpcSetInterrupt()` function.

To delete an IPC synchronization object, close its handle by calling the `KnHandleClose()` function that is declared in the header file `sysroot-*-kos/include/coresrv/handle/handle_api.h` from the KasperskyOS SDK.

Information about API functions

ipc_api.h functions

Function	Information about the function
`KnIpcCreateSyncObject()`	Purpose Creates an IPC synchronization object. Parameters • [out] `syncHandle` – pointer to the handle of the IPC synchronization object. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnIpcSetInterrupt()`	Purpose

	Switches the defined IPC synchronization object to a state in which the `Call()` and `Recv()` system calls are interrupted. Parameters • [in] `syncHandle` – handle of the IPC synchronization object. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnIpcClearInterrupt()`	Purpose Switches the defined IPC synchronization object to a state in which the `Call()` and `Recv()` system calls are not interrupted. Parameters • [in] `syncHandle` – handle of the IPC synchronization object. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Using sessions (session.h)

The API is defined in the header file `sysroot-*-kos/include/kos/session.h` from the KasperskyOS SDK.

The API is intended for use in the code of resource providers that provide access to user resources via IPC, and in the code of programs that utilize user resources locally (without IPC). The API lets you organize session-based management of access to user resources. A session consists of a sequence of actions, such as opening access to a resource, using the resource, and closing access to the resource. When access to a resource is opened, a session is created. Closing access to the resource terminates the session. Multiple operations can be simultaneously performed with a resource in one session. A resource can be used in exclusive mode or multi-access mode. In the latter case, the resource must be opened multiple times (possibly with different access permissions). In other words, the resource will be used via multiple, simultaneously existing sessions (parallel sessions).

In the API description, an "object" refers to a `KosObject`, which is an abstraction of a resource (for details about `KosObjects`, see Using KosObjects (objects.h)). The type of session and session context are also `KosObjects`. (A session context contains information about this session, such as the resource access permissions and the counter for active operations with the resource.)

Information about API functions is provided in the table below.

Creating a session type

A session type is created before sessions are created. To create a session type, call the `KosSessionCreateType()` function.

You must use the `objectType` parameter to define the type of objects for which the created type of sessions are intended. The object type must inherit the type with the ID `kosSessionBaseObjectType`.

Use the `sessionType` parameter to define the type of session contexts. You can specify the base type of session contexts with the ID `kosSessionBaseSessionType` or the type that inherits it. The base type of session contexts corresponds to the structure containing data such as the pointer to the object, the resource access permissions, the resource handle permissions mask that is transferred to resource consumers, and the counter of active operations with the resource during the session (the type of this structure is not exported to programs, but instead is used by the `libkos` library). You can add additional information to the session context by creating your own session context type that inherits the base type. For example, the session context for device usage may additionally contain pointers to functions that are called by solution programs to work with the device.

You must use the `ops` parameter to define the callback functions that are called when creating and terminating sessions. The callback functions for the `open` and `close` IDs must be defined. You do not have to define the callback function for the `closeComplete` ID, therefore this ID can be set to `RTL_NULL`.

The callback function corresponding to the `open` ID is called when creating a session and receives the pointers to the object, session context, data passed to the `KosSessionOpenObject()` function via the `openCtx` parameter, and the buffer for saving the resource handle permissions mask transferred to resource consumers. In this callback function, you can check the status of the resource, define the resource handle permissions mask transferred to resource consumers, and write data to the object or session context, for example. If the return code is different from `rcOk`, a session will not be created.

The callback function corresponding to the `close` ID is called when the session is terminated, irrespective of whether operations with the resource were completed (the counter of active operations in the closed session may be greater than zero), and receives pointers to the object and session context. In this callback function, you can interrupt active operations in a terminated session and write data to an object or session context, for example. Any return code is acceptable.

The callback function corresponding to the `closeComplete` ID is called when a session is terminated only after completion of all operations with the resource (the counter of active operations in the completed session is equal to zero) and receives the pointers to the object and session context. In this callback function, you can write data to an object, for example.

Registering an object

After creating a session type, you must register the required objects as objects that can be linked to sessions of this type. To register an object, call the `KosSessionRegisterObject()` function.

You cannot register one object to be linked to different types of sessions at the same time.

After registration, the object is ready to be opened.

Opening an object

When an object is opened, a session linked to this object is created, and an object is opened when there is a request to open access to a resource. This may be a request via IPC from the resource consumer or a local request from another program component.

To open an object, call the `KosSessionOpenObject()` function. You must use the `requiredRights` parameter to define the resource access permissions in the created session, and use the `shareMode` parameter to define the multi-access permissions for a resource to properly restrict access to this resource for parallel sessions. For example, if read-and-write permissions are defined via the `requiredRights` parameter and the `shareMode` parameter is set to `0`, the created session requires exclusive read-and-write access to the resource. If read-and-write permissions are defined via the `requiredRights` parameter but only read permissions are defined via the `shareMode` parameter, the created session requires read-and-write permissions and allows read operations in parallel sessions. If read permissions are defined via the `requiredRights` parameter but read-and-write permissions are defined via the `shareMode` parameter, the created session requires read-access and allows read-and-write operations in parallel sessions.

If a resource needs to be used in multi-access mode, you must consider the following conditions when the object is opened a second time and subsequent times:

- The access permissions defined via the `requiredRights` parameter cannot exceed the minimal access permissions defined via the `shareMode` parameter for all parallel sessions linked to this object.

- The access permissions defined via the `shareMode` parameter must include all of the access permissions defined via the `requiredRights` parameter for all parallel sessions linked to this object.

The session context will be created after the object is opened.

If a program uses resources locally, after the object is opened, the program component managing the resources must pass the pointer of the created session context to the program component that uses the resources. When subsequently requesting operations with a resource, the program component using the resources must pass this pointer to the program component managing the resources for session identification.

After opening an object, the resource provider must transfer the resource handle to the consumer. The session context must be used as the resource transfer context so that the consumer can receive the handle of the resource linked to the session. When subsequently requesting operations with a resource, the consumer must put the received resource handle into IPC requests so that the provider receives the pointer to the session context after dereferencing this handle. To fill the resource handle transport container with data for transfer to a resource consumer, call the `KosSessionContextToIpcHandle()` function. This function receives the session context via the `sessionCtx` parameter and transfers the resource handle transport container via the `desc` parameter.

The `KosSessionContextToIpcHandle()` function takes the permissions mask defined in the callback function that is called when creating the session and puts this permissions mask into the resource handle transport container. The access permissions defined in this mask must not exceed the session access permissions defined via the `requiredRights` parameter of the `KosSessionOpenObject()` function. However, the same access permission may correspond to different bits in the permissions mask and in the value defined via the `requiredRights` parameter of the `KosSessionOpenObject()` function.

The `KosSessionContextToIpcHandle()` function defines the session context as the resource transfer context. The session context will be automatically deleted (and therefore the session will be terminated) after closure or revocation of the resource handles that were generated by the transfer of the resource handle that was received by calling the `KosSessionContextToIpcHandle()` function. If you need to prevent automatic termination of a session for subsequent termination via the `KosSessionCloseObject()` function call, prior to transferring the resource handle you must increment the counter for links to the context of this session by calling the `KosRefObject()` function from the objects.h API. In this case, after calling the `KosSessionCloseObject()` function, you must decrement the counter for links to the session context by calling the `KosPutObject()` function from the objects.h API.

Performing operations with a resource

An operation with a resource (for example, reading, writing, or getting parameters) is performed by request via IPC from the resource consumer or by local request from another program component.

The scenario for performing an operation with a resource includes the followings steps:

1. Verify the resource access permissions.

   When processing a local request, call the `KosSessionGetOpenRights()` function. This function gets information about session resource access permissions defined via the `requiredRights` parameter of the `KosSessionOpenObject()` function to compare these permissions with the permissions needed to perform the operation.

   During IPC request processing, you must call the `KosSessionIpcHandleToSession()` function, which gets the session context and verifies whether the permissions mask of the resource handle that the resource consumer put into the IPC request corresponds to the requested operation. The resource handle transport container received from the resource consumer must be passed to the function via the `desc` parameter. In the `operation` parameter, you must specify the resource access permissions that are needed to perform the requested operation. In the `type` parameter, you must specify the ID of the session type to verify that the type of session context matches the session type. If you set the `type` parameter to `RTL_NULL`, you must verify the type of session context prior to converting the pointer type that was received via the `sessionCtx` parameter. Verification of the session context type is mandatory because the resource provider may create sessions with contexts of different types while the resource consumer may erroneously make an IPC request with the resource handle matching a session with a different type of context. To verify the context type, use the objects.h API.

2. Increment the counter for active operations in the session by calling the `KosSessionRef()` function.

   This function verifies that the session has not terminated. In the `type` parameter, you must specify the ID of the session type to verify that the type of session context matches the session type. You can use the `object` parameter to receive a pointer to the object.

3. Perform the requested actions with the resource.

4. Decrement the counter for active operations in the session by calling the `KosSessionPut()` function.

Getting a resource handle

A resource handle may be required for using notifications or for querying the Kaspersky Security Module through the security interface, for example. To get a resource handle for an object or session context, call the `KosSessionGetObjectHandle()` or `KosSessionGetObjectHandleBySession()` function, respectively.

Enumerating sessions linked to an object

You may need to enumerate sessions linked to an object to notify each of the resource consumers using a resource about the state of that resource, for example. To enumerate sessions linked to an object, call the `KosSessionWalk()` function. Use the `handler` parameter to define the callback function that is called for each session during enumeration and receives pointers to the object, session context and data that is passed to the `KosSessionWalk()` function via the `walkCtx` parameter. In this callback function, you can send a notification about the state of a resource to consumers of that resource, for example.

Closing an object

When an object is closed, one of the sessions linked to this object is terminated. A session will not be terminated until all operations in this session are terminated (in other words, when the counter of active operations is equal to zero). When a session is terminated, the handles of the resource linked to this session are automatically revoked.

An object is closed when there is a request to close access to the resource. This may be a request via IPC from the resource consumer or a local request from another program component. After making a request to close access to a resource, the resource consumer must close the handle of this resource. (If the resource consumer terminates without making a request to close access to the resource, or closes the resource handle without making this request, the corresponding session will be automatically terminated under the condition that the resource consumer has not additionally incremented the counter for links to the context of this session.)

To close an object, call the `KosSessionCloseObject()` or `KosSessionCloseObjectByIpcHandle()` function.

The `KosSessionCloseObject()` function terminates the session corresponding to the session context that was defined via the `sessionCtx` parameter.

The `KosSessionCloseObjectByIpcHandle()` function terminates the session linked to the resource handle that the resource consumer put into the IPC request to close access to the resource. The received resource handle transport container must be passed to the function via the `desc` parameter. In the `type` parameter, you must specify the ID of the session type to verify that the type of session context matches the session type.

Deregistering an object

When an object is deregistered, all sessions linked to this object are terminated and it is no longer possible to open this object. A session will not be terminated until all operations in this session are terminated (in other words, when the counter of active operations is equal to zero). When a session is terminated, the handles of the resource linked to this session are automatically revoked. (Revocation does not close resource handles, therefore revoked handles need to be closed.)

To deregister an object, call the `KosSessionUnregisterObject()` function.

Deleting a session type

A session type must be deleted if there are no registered objects for which sessions of this type can be used, and there will be no more registrations. For example, a session type must be deleted when the resource provider is terminated.

To delete a session type, call the `KosSessionDestroyType()` function.

Information about API functions

session.h functions

Function	Information about the function
`KosSessionCreateType()`	Purpose Creates a session type. Parameters • [in] `objectType` – ID of the object type for which the sessions are intended. The parameter type is defined in the header file `sysroot-*-kos/include/kos/objects.h` from the KasperskyOS SDK. • [in] `sessionType` – ID of the session context type. The parameter type is defined in the header file `sysroot-*-kos/include/kos/objects.h` from the KasperskyOS SDK. • [in] `ops` – pointer to the structure containing the IDs of callback functions that are called when sessions are created and terminated. • [out] `outType` – pointer to the ID of the session type.

	Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosSessionDestroyType()`	Purpose Deletes a session type. Parameters • [in] `type` – ID of the session type. Returned values N/A
`KosSessionRegisterObject()`	Purpose Registers an object as an object that can be linked to sessions of the defined type. Parameters • [in] `type` – ID of the session type. • [in] `object` – pointer to the object. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosSessionUnregisterObject()`	Purpose Deregisters an object that was registered by calling the `KosSessionRegisterObject()` function. Parameters • [in] `object` – pointer to the object. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosSessionOpenObject()`	Purpose Opens an object. Parameters • [in] `object` – pointer to the object. • [in,optional] `openCtx` – pointer to the data that will be passed to the callback function that is called when the session is created. You can specify `RTL_NULL` if data does not need to be passed. The callback function will receive the passed data through its own `ctx` parameter. • [in] `requiredRights` – value that defines the resource access permissions in a created session. • [in] `shareMode` – value that defines multi-access permissions for a resource. • [out] `sessionCtx` – pointer to the session context address. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosSessionCloseObject()`	Purpose Closes an object. Parameters • [in] `sessionCtx` – pointer to the session context.

	Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosSessionRef()`	Purpose Increments the counter for active operations in a session. Parameters • [in,optional] `type` – ID of the session type, or `RTL_NULL` if there is no need to verify that the session context type matches the session type. • [in] `sessionCtx` – pointer to the session context. • [out,optional] `object` – pointer to the object address, or `RTL_NULL` if the object address is not required. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosSessionPut()`	Purpose Decrements the counter for active operations in a session. Parameters • [in] `sessionCtx` – pointer to the session context. Returned values N/A
`KosSessionWalk()`	Purpose Enumerates the sessions linked to a defined object. Parameters • [in] `object` – pointer to the object. • [in] `handler` – ID of the callback function that is called for each session during enumeration. • [in,optional] `walkCtx` – pointer to the data that will be passed to the callback function that was defined via the `handler` parameter. You can specify `RTL_NULL` if data does not need to be passed. The callback function will receive the passed data through its own `walkCtx` parameter. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information If the callback function defined via the `handler` parameter returns `rtl_false`, enumeration is terminated. The callback function defined via the `handler` parameter must not call the `KosSessionOpenObject()`, `KosSessionCloseObject()`, `KosSessionRegisterObject()`, and `KosSessionUnregisterObject()` functions.
`KosSessionGetOpenRights()`	Purpose Gets information about resource access permissions in the specified session. These permissions are defined via the `requiredRights` parameter of the `KosSessionOpenObject()` function. Parameters • [in] `sessionCtx` – pointer to the session context.

	Returned values If successful, the function returns the value indicating the resource access permissions, otherwise it returns `0`.
`KosSessionContextToIpcHandle()`	Purpose Fills the resource handle transport container with data for transfer to a resource consumer when processing an IPC request to open access to the resource. Parameters • [in] `sessionCtx` – pointer to the session context. • [out] `desc` – pointer to the handle transport container. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosSessionIpcHandleToSession()`	Purpose Gets the session context when processing an IPC request from a resource consumer. Parameters • [in] `desc` – pointer to the handle transport container. • [in] `operation` – value that indicates the resource access permissions required for performing operations to verify that the handle permissions mask put into the IPC request by the resource consumer matches the requested operation. • [in,optional] `type` – ID of the session type, or `RTL_NULL` if there is no need to verify that the session context type matches the session type. • [out] `sessionCtx` – pointer to the session context address. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosSessionCloseObjectByIpcHandle()`	Purpose Closes an object when processing an IPC request from a resource consumer to close access to the resource. Parameters • [in] `desc` – pointer to the handle transport container. • [in,optional] `type` – ID of the session type, or `RTL_NULL` if there is no need to verify that the session context type matches the session type. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosSessionGetObjectHandle()`	Purpose Gets a resource handle for an object. Parameters • [in] `object` – pointer to the object. • [out] `handle` – pointer to the handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosSessionGetObjectHandleBySession()`	Purpose Gets a resource handle for a session context.

	Parameters • [in] `sessionCtx` – pointer to the session context. • [out] `handle` – pointer to the handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Using KosObjects (objects.h)

The API is defined in the header file `sysroot-*-kos/include/kos/objects.h` from the KasperskyOS SDK.

The API lets you use typified `KosObjects`. A `KosObject` is a process memory region consisting of segments containing service data and payload data. Service data includes a pointer to the structure describing the object type, a counter for links to the object, and other information about the object. Payload data may consist of any data depending on the purpose of the object. For example, segments of payload data in an object representing a device abstraction save the structures that contain the device operating parameters, pointers to low-level methods for working with the device, and the parameters of software/hardware interfaces for device management.

A type inheritance mechanism is supported for `KosObjects`. For example, an abstraction of the NS16550 UART device contains the following structures in different segments of payload data: a structure specific to NS16550 UART devices, a structure specific to the UART device class, and common structures for all devices. The NS16550 UART device abstraction type corresponds only to the structure that is specific to NS16550 UART devices. All other structures correspond to types that inherited the NS16550 UART device abstraction type.

In the API, the pointer to an object is the pointer to the segment of payload data in this object.

Information about API functions is provided in the table below.

Creating an object type

An object type is created before objects are created. An object type is a process memory region containing the structure that includes data such as the type name, pointers to functions that are called when objects of this type are created and destroyed, a pointer to the parent type, and a counter for links to the type. Each object stores the pointer to this structure, which means that the objects are typified.

To create an object type, call the `KosCreateObjectType()` or `KosCreateObjectTypeEx()` function. In contrast to the `KosCreateObjectType()` function, the `KosCreateObjectTypeEx()` function lets you create a type that inherits other types. This type contains the address of the parent type that is defined via the `parentType` parameter. You can also use the `parentType` parameter to define a type that also contains the address of its parent type. As a result, you can use the `KosCreateObjectTypeEx()` function to create a type inheritance tree. Each type in this tree will inherit the entire chain of types that form the path from the root of the tree to this specific type.

Depending on whether type inheritance is being employed, objects may be simple or composite. Objects that correspond to one type are known as simple objects. Objects that correspond to a type inheritance chain are known as composite objects. Composite ones consist of simple objects. The number of simple objects in a composite object is equal to the number of types in the inheritance chain. The simple objects form an inheritance chain corresponding to the chain of inheritance of their types. In other words, each object except the first one in the inheritance chain contains the address of the parent object.

You need to use the `defaultObjSize` parameter of the `KosCreateObjectType()` and `KosCreateObjectTypeEx()` functions to define the default size of objects (you can define a different size when creating an object). When creating a type that inherits other types, this parameter pertains only to the last type in the inheritance chain. The `defaultObjSize` parameter effectively defines the minimum size of a segment of payload data in simple objects because the actual size of objects will be larger than the defined size due to the alignment of memory regions, inheritance of types, and the presence of segments containing service data.

You can use the `ops` parameter of the `KosCreateObjectType()` and `KosCreateObjectTypeEx()` functions to define the callback functions that are called when objects are created and destroyed (you can define both functions or just one of them). When a composite object is created, a set of simple objects is created within it. When the composite object is destroyed, the set of simple objects within it is also destroyed. When each simple object is created and destroyed, separate callback functions that were defined when creating the type of this object are called. The first object in the inheritance chain is created first, and the last object in the inheritance chain is created last. The last object in the inheritance chain is destroyed first, and the first object in the inheritance chain is destroyed last.

The callback function that is called when creating an object receives pointers to the object and the data passed to the `KosCreateObjectEx()` function via the `context` parameter. In this callback function, you can allocate resources and initialize data in an object, for example. If the return code differs from `rcOk`, the object will not be created. If a composite object is being created without creating at least one simple object, this composite object will not be created. Also, when creating a composite object, the data passed to the `KosCreateObjectEx()` function via the `context` parameter will be received only by the callback function that is called when creating the last simple object in the inheritance chain.

The callback function that is called when an object is destroyed receives the pointer to the object. In this callback function, you can free up resources, for example. Any return code is acceptable.

Creating an object

After creating an object type, you can create objects. To create an object, you must call the `KosCreateObject()` or `KosCreateObjectEx()` function.

Use the `type` parameter to define the object type. If you define a type that does not inherit other types, a simple object will be created. If you define a type that inherits other types, a composite object will be created and the last simple object in the inheritance chain for this composite object will have the defined type.

Use the `size` parameter to define the object size, or specify `0` to set the default object size specified in the type of this object. If a composite object is created, this parameter pertains only to the last simple object in the inheritance chain. The `size` parameter effectively defines the minimum size of a segment of payload data in a simple object because the actual size of the object will be larger than the defined size due to the alignment of the memory region, inheritance of types, and the presence of segments containing service data.

In contrast to the `KosCreateObject()` function, the `KosCreateObjectEx()` function has the `context` parameter, which can be used to pass data to the callback function that is called when an object is created. When creating a composite object, the data passed via the `context` parameter will be received only by the callback function that is called when creating the last simple object in the inheritance chain.

The `outObject` parameter is used by the `KosCreateObject()` and `KosCreateObjectEx()` functions to transfer the pointer of the created object. The pointer to the created composite object is the pointer to the last simple object in the inheritance chain.

After it is created, the object can be used to write data to it and read data from it.

Obtaining access to simple objects in a composite object

To gain access to simple objects in a composite object, you need to use the `KosGetObjectParent()` function and/or the `KosGetObjectAncestor()` function. The `KosGetObjectParent()` function gets the pointer to the object that is the direct ancestor of the defined object. The `KosGetObjectAncestor()` function gets the pointer to the object that is the ancestor of the defined object and has the defined type.

Managing the lifetime of objects and their types

An object exists as long as the counter for links to that object has a value greater than zero. Likewise, an object type exists as long as the counter for links to that object has a value greater than zero. When an object or object type is created, the value of the links counter is equal to 1. This counter can then be incremented or decremented to manage the lifetime of the object or object type. For example, after an object is created, you can decrement the counter for links to its type because the `libkos` library increments the counter for links to this object type when an object is created. In this case, the object type will be destroyed automatically after the object is destroyed because the `libkos` library decrements the counter for links to the object type when an object is destroyed. In addition, when saving an object address in another object, you must increment the counter for links to the first object to ensure that it exists throughout the lifetime of the second object. When you destroy an object that contains the address of another object, you must decrement the counter for links to the second object to ensure that it is destroyed when no other links remain.

The lifetime of a simple object within a composite object corresponds to the lifetime of this composite object. You cannot separately manage the lifetime of a simple object within a composite object.

To increment the counter for links to an object or object type, call the `KosRefObject()` or `KosRefObjectType()` function, respectively.

To decrement the counter for links to an object or object type, call the `KosPutObject()` or `KosPutObjectType()` function, respectively.

Verifying an object type

The `KosCheckParentType()` and `KosObjectTypeInheritsType()` functions let you check whether one object type is a parent to another object type.

The `KosObjectOfType()` function verifies whether an object has the defined type.

The `KosObjectInheritsType()` function checks whether an object has the defined type or a type inherited by the defined type.

Getting the type and name of an object

To get the object type, call the `KosObjectGetType()` function.

To get the object name, call the `KosGetObjectName()` function. (Each simple object in a composite object has the name of this composite object.)

Getting the name of an object type

To get the name of an object type, call the `KosObjectGetTypeName()` function.

Information about API functions

Function	Information about the function
`KosCreateObjectType()`	Purpose Creates an object type. Parameters • [in] `name` – pointer to the name of the object type. • [in] `defaultObjSize` – default size of objects, in bytes. • [in,optional] `ops` – pointer to the structure containing pointers to the callback functions that are called when creating and destroying objects. You can set this to `RTL_NULL` if you do not need to define these callback functions. • [out] `outType` – pointer to the ID of the object type. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosCreateObjectTypeEx()`	Purpose Creates an object type. Parameters • [in] `name` – pointer to the name of the object type. • [in] `id` – fictitious parameter. You must specify `0`. • [in] `defaultObjSize` – default size of objects, in bytes. • [in,optional] `ops` – pointer to the structure containing pointers to the callback functions that are called when creating and destroying objects. You can set this to `RTL_NULL` if you do not need to define these callback functions. • [in,optional] `parentType` – ID of the parent object type, or `RTL_NULL` if inheritance of other object types is not required. • [out] `outType` – pointer to the ID of the object type. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosCreateObjectEx()`	Purpose Creates an object. Parameters • [in] `type` – ID of the object type. • [in,optional] `name` – pointer to the object name, or `RTL_NULL` if you do not need to define the object name. • [in,optional] `size` – size of the object (in bytes), or `0` to set the default object size. • [in,optional] `context` – pointer to the data that will be passed to the callback function that is called when the object is created. You can specify `RTL_NULL` if data does not need to be passed. The callback function will receive the passed data through its own `context` parameter. • [out] `outObject` – pointer to the object address. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information

	When an object is created, memory is allocated for that object and this memory is filled with zeros.
`KosRefObjectType()`	Purpose Increments the counter for links to an object type. Parameters • [in] `type` – ID of the object type. Returned values N/A
`KosPutObjectType()`	Purpose Decrements the counter for links to an object type. Parameters • [in] `type` – ID of the object type. Returned values N/A
`KosCheckParentType()`	Purpose Verifies that the specific object `type` is a direct descendant of the `parentType` object type. Parameters • [in] `type` – ID of the object type. • [in] `parentType` – ID of the object type. Returned values If the verification is successful, the function returns `rtl_true`, otherwise it returns `rtl_false`.
`KosObjectTypeInheritsType()`	Purpose Verifies that the specific object `type` is a descendant of the `parentType` object type. Parameters • [in] `type` – ID of the object type. • [in] `parentType` – ID of the object type. Returned values If the verification is successful, the function returns `rtl_true`, otherwise it returns `rtl_false`.
`KosCreateObject()`	Purpose Creates an object. Parameters • [in] `type` – ID of the object type. • [in,optional] `name` – pointer to the object name, or `RTL_NULL` if you do not need to define the object name. • [in,optional] `size` – size of the object (in bytes), or `0` to set the default object size. • [out] `outObject` – pointer to the object address. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information

	When an object is created, memory is allocated for that object and this memory is filled with zeros.
`KosRefObject()`	Purpose Increments the counter for links to an object. Parameters • [in] `object` – pointer to the object. Returned values N/A
`KosPutObject()`	Purpose Decrements the counter for links to an object. Parameters • [in] `object` – pointer to the object. Returned values N/A
`KosGetObjectName()`	Purpose Gets the object name. Parameters • [in] `object` – pointer to the object. Returned values Returns the ID of the `KosString` containing the object name. The type of returned value is defined in the header file `sysroot-*-kos/include/kos/strings.h` from the KasperskyOS SDK. Additional information The `KosGetObjectName()` function increments the counter for links to the `KosString` containing the object name. If this string is no longer needed, you need to decrement the number of links to it by calling the `KosPutString()` function from the strings.h API.
`KosGetObjectParent()`	Purpose Gets the pointer to the object that is the direct ancestor of the defined object. Parameters • [in] `object` – pointer to the object. Returned values If successful, the function returns the pointer for the object that is the direct ancestor of the defined object, otherwise it returns `RTL_NULL`.
`KosGetObjectAncestor()`	Purpose Gets the pointer to the object that is the ancestor of the defined object and has the defined type. Parameters • [in] `object` – pointer to the object. • [in] `type` – ID of the object type. Returned values If successful, the function returns the pointer for the object that is the ancestor of the defined object and has the defined type, otherwise it returns `RTL_NULL`. Additional information

	If the `object` parameter is used to define an object that has the type defined via the `type` parameter, the function returns the pointer for this object.
`KosObjectOfType()`	Purpose Verifies that the object has the defined type. Parameters • [in] `object` – pointer to the object. • [in] `type` – ID of the object type. Returned values If the verification is successful, the function returns `rtl_true`, otherwise it returns `rtl_false`.
`KosObjectInheritsType()`	Purpose Verifies that the object has the defined type or the type that inherits the defined type. Parameters • [in] `object` – pointer to the object. • [in] `type` – ID of the object type. Returned values If the verification is successful, the function returns `rtl_true`, otherwise it returns `rtl_false`.
`KosObjectGetType()`	Purpose Gets the ID of the object type. Parameters • [in] `object` – pointer to the object. Returned values Object type ID. Additional information Increments the counter for links to an object type. If the received object type ID is no longer needed, you must decrement the number of links to the object type by calling the `KosPutObjectType()` function.
`KosObjectGetTypeName()`	Purpose Gets the name of the object type. Parameters • [in] `type` – ID of the object type. Returned values Returns the ID of the `KosString` containing the object type name. The type of returned value is defined in the header file `sysroot-*-kos/include/kos/strings.h` from the KasperskyOS SDK. Additional information The `KosObjectGetTypeName()` function increments the counter for links to the `KosString` containing the object type name. If this string is no longer needed, you need to decrement the number of links to it by calling the `KosPutString()` function from the strings.h API.

Using KosObject containers (objcontainer.h)

The API is defined in the header file `sysroot-*-kos/include/kos/objcontainer.h` from the KasperskyOS SDK.

The API lets you merge `KosObjects` into containers to conveniently use sets of these objects (for details about `KosObjects`, see Using KosObjects (objects.h)). The containers are also `KosObjects` and may be elements of other containers. The same `KosObject` may be an element of multiple containers at the same time.

Information about API functions is provided in the table below.

Creating a container

To create a container, call the `KosCreateObjContainer()` function. In the `parent` parameter, you can specify the ID of the parent container, which is the container to which the created container will be added.

Adding an object to a container

To add an object to a container, call the `KosInsertObjContainerObject()` function. An object can be another container. You must use the `name` parameter to define the object name that this object will have within the container. This name is not related to the name that was defined when the object was created. The name of an object within a container must be unique so that this object can be unambiguously identified among other objects in this container. When an object is added to a container, the counter for links to this object is incremented.

Removing an object from a container

To remove an object from a container, call the `KosRemoveObjContainerObjectByName()` or `KosRemoveObjContainerObject()` function. When an object is removed from a container, the counter for links to this object is decremented.

Searching for an object in a container

To search for an object with the defined name in a container, call the `KosGetObjContainerObjectByName()` function. Child containers will not be searched for the object. This function increments the counter for links to the found object.

Enumerating objects in a container

Enumeration may be necessary to perform specific actions with multiple objects in a container. To enumerate objects in a container, call the `KosWalkObjContainerObjects()` function. Use the `walk` parameter to define the callback function that is called for each object during enumeration and receives pointers to the object and data that is passed to the `KosWalkObjContainerObjects()` function via the `context` parameter. Objects in child containers are not enumerated.

Enumerating the names of objects in a container

To enumerate the names of objects in a container, use the `KosEnumObjContainerNames()` function. The order of enumeration for object names matches the order in which these objects are added to the container. Object names in child containers are not enumerated.

Getting the number of objects in a container

To get the number of objects in a container, call the `KosCountObjContainerObjects()` function. Objects in child containers are not taken into account.

Clearing a container

To remove all objects from a container, call the `KosClearObjContainer()` function. This function decrements the counters for links to objects that are removed from the container.

Checking whether an object is a container

To check whether an object is a container, call the `KosIsContainer()` function.

Deleting a container

To delete a container, call the `KosDestroyObjContainer()` function. When a container is deleted, the counters for links to objects within this container are decremented.

Information about API functions

objcontainer.h functions

Function	Information about the function
`KosCreateObjContainer()`	Purpose Creates a container. Parameters • [in,optional] `parent` – ID of the parent container, or `RTL_NULL` if you do not need to add the created container to an already existing container. • [in] `name` – pointer to the name of the created container. • [out] `outContainer` – ID of the created container. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosDestroyObjContainer()`	Purpose Deletes a container. Parameters • [in] `container` – ID of the container. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosInsertObjContainerObject()`	Purpose Adds an object to a container. Parameters • [in] `container` – ID of the container.

	• [in] `name` – pointer to the object name. • [in] `object` – pointer to the object. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosRemoveObjContainerObjectByName()`	Purpose Removes the object with the defined name from a container. Parameters • [in] `container` – ID of the container. • [in] `name` – pointer to the object name. • [in,optional] `object` – pointer to the object, or `RTL_NULL` if you do not need to verify that the object name defined via the `name` parameter matches the object being removed. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosRemoveObjContainerObject()`	Purpose Removes an object from a container. Parameters • [in] `container` – ID of the container. • [in] `object` – pointer to the object. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosGetObjContainerObjectByName()`	Purpose Searches a container for the object with the defined name. Parameters • [in] `container` – ID of the container. • [in] `name` – pointer to the name of the object that you are searching for. • [out] `outObject` – pointer to the address of the found object. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information If an object with the defined name is not found, the function returns the `RTL_NULL` value via the `outObject` parameter.
`KosWalkObjContainerObjects()`	Purpose Enumerates objects in a container and calls the defined function for each object during enumeration. Parameters • [in] `container` – ID of the container. • [in] `walk` – ID of the callback function that is called for each object during enumeration. The parameter type is defined in the header file `sysroot-*-kos/include/kos/objects.h` from the KasperskyOS SDK.

	• [in,optional] `context` – pointer to the data that will be passed to the callback function that was defined via the `walk` parameter. You can specify `RTL_NULL` if data does not need to be passed. The callback function will receive this data through its own `context` parameter. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information The callback function defined via the `walk` parameter must not call other API functions. If a callback function defined via the `walk` parameter returns a value with the `KOS_OBJECT_WALK_FINISH` flag, the `KosWalkObjContainerObjects()` function terminates enumeration. If a callback function defined via the `walk` parameter returns a value with the `KOS_OBJECT_WALK_REMOVE_OBJECT` flag, the `KosWalkObjContainerObjects()` function removes the corresponding object from the container. The `KOS_OBJECT_WALK_FINISH` and `KOS_OBJECT_WALK_REMOVE_OBJECT` flags are defined in the header file `sysroot-*-kos/include/kos/objects.h` from the KasperskyOS SDK.
`KosEnumObjContainerNames()`	Purpose Enumerates the names of objects in a container. Parameters • [in] `container` – ID of the container. • [in] `index` – index for enumerating object names. Enumeration starts with zero. • [out] `outStr` – pointer to the ID of the `KosString` containing the object name. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. Additional information The `KosEnumObjContainerNames()` function increments the counter for links to the `KosString` containing the object name. If this string is no longer needed, you need to decrement the number of links to it by calling the `KosPutString()` function from the strings.h API.
`KosCountObjContainerObjects()`	Purpose Gets the number of objects in a container. Parameters • [in] `container` – ID of the container. Returned values Number of objects in a container.
`KosClearObjContainer()`	Purpose Clears a container. Parameters • [in] `container` – ID of the container. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosIsContainer()`	Purpose Checks whether an object is a container.

	Parameters • [in] `object` – pointer to the object. Returned values If the verification is successful, the function returns `rtl_true`, otherwise it returns `rtl_false`.

Using KosStrings (strings.h)

The API is defined in the header file `sysroot-*-kos/include/kos/strings.h` from the KasperskyOS SDK.

The API lets you use `KosStrings`, which have the following distinguishing characteristics:

1. A `KosString` is a C-string (with a terminating zero) that follows the header containing the service data.

2. `KosStrings` are stored in hash tables, which are arrays whose elements are lists of strings. (The pointers for creating linked lists of strings are stored in the headers of strings.) Each list consists of one or more strings. Each list contains the strings who have the same value for the remainder after dividing the hash value of the string by the number of lists in the hash table. The main property of a hash table is that the computational complexity of searching for a string does not depend on the degree of occupancy of this table if all lists include only one string.

   When adding `KosStrings` to a hash table in real-world scenarios, lists containing multiple strings may be created. This leads to varying computational complexity when searching for strings in the hash table for the following reason. When searching for a key, the hash value is calculated and then converted into a string list index (by calculating the remainder from dividing the hash value by the number of lists). If there is only one string in the list containing the calculated index, the search will be complete after comparing this string to the key. However, if there are multiple strings in the list containing the calculated index, you may need to compare the key to all strings in this list to complete the search. In this case, the computational complexity of searching for a string may be higher than when there is only one string in the list.

3. `KosStrings` from one hash table are unique, which means that there are no identical strings in a hash table. If the strings from one hash table have different IDs, that means that these are different strings. (In the API, the ID of the `KosString` is the pointer to this string.) This lets you compare strings based on their IDs instead of their contents, thereby ensuring that the computational complexity of the comparison does not depend on the sizes of strings.

4. The size of a `KosString` is stored in the header instead of being calculated based on a search for a terminating zero, thereby ensuring that the computational complexity of getting the size does not depend on the actual size of the string.

5. `KosStrings` cannot be modified. This characteristic is a result of characteristics 2–4. Modification of a string may lead to the appearance of identical strings in the same hash table, a discrepancy between the actual size of a string and the size stored in the header, and a discrepancy between the string contents and the index of the string list in which this string is listed in the hash table.

6. A `KosString` exists as long as the counter for links to it remains greater than zero. (The string links counter is stored in the header.)

Information about API functions is provided in the table below.

Creating a hash table

You can use the default hash table, which is created automatically during initialization of the `libkos` library. This table may include no more than 2039 lists of strings with a string size of no more than 65,524 bytes without taking into account the terminating zero.

You can create a hash table with the required maximum number of string lists and the required maximum size of strings by calling the `KosCreateStringRoot()` function.

A hash table is created empty without any strings.

Adding a KosString to a hash table

To add a string to the default hash table, call the `KosCreateString()` function.

To add a string to a defined hash table, call the `KosCreateStringEx()` function.

If the string being added is already listed in the hash table, these functions do not add a new string but instead pass the ID of the already existing string via the `outStr` parameter.

Searching for a KosString in a hash table

To search for a string in the default hash table, call the `KosGetString()` function.

To search for a string in a defined hash table, call the `KosGetStringEx()` function.

The search is completed successfully if the key completely matches the string.

Managing the lifetime of KosStrings

A string exists as long as the counter for links to it remains greater than zero. The `KosCreateString()` and `KosCreateStringEx()` functions add a string whose links counter has a value of 1 to the hash table. If the string has already been added to the hash table, these functions increment the counter for links to this string. The `KosGetString()` and `KosGetStringEx()` functions increment the counter for links to the found string. This counter can then be incremented or decremented to manage the lifetime of the string. When passing the ID of a string to another program component, you must increment the counter for links to the corresponding string to ensure that this string will exist for the amount of time required by this component. If the string is no longer needed, you must decrement the counter for links to this string to ensure that it is destroyed when no other links remain.

To increment the counter for links to a string, call the `KosRefString()` or `KosRefStringEx()` function.

To decrement the counter for links to a string in the default hash table, call the `KosPutString()` function.

To decrement the counter for links to a string in a defined hash table, call the `KosPutStringEx()` function.

Getting the size of a KosString

To get the size of a string without taking into account the terminating zero, call the `KosGetStringLen()` function.

To get the size of a string while taking into account the terminating zero, call the `KosGetStringSize()` function.

Deleting a hash table

To delete a hash table, call the `KosDestroyStringRoot()` function. The table will be deleted only if it is empty.

Information about API functions

strings.h functions

Function	Information about the function
`KosCreateStringRoot()`	Purpose Creates an empty hash table. Parameters • [in] `numBuckets` – maximum number of `KosString` lists. • [in] `maxStrLength` – maximum size of `KosStrings` without taking into account the terminating zero, in bytes. Returned values ID of the hash table.
`KosDestroyStringRoot()`	Purpose Deletes an empty hash table. Parameters • [in] `handle` – ID of the hash table. Returned values N/A
`KosGetString()`	Purpose Searches for a `KosString` in the default hash table. Parameters • [in] `str` – pointer to the C-string containing the search key. Returned values If successful, the function returns the ID of the `KosString`, otherwise it returns `RTL_NULL`.
`KosGetStringEx()`	Purpose Searches for a `KosString` in a defined hash table. Parameters • [in,optional] `root` – ID of the hash table, or `RTL_NULL` to assign the default hash table. • [in] `str` – pointer to the C-string containing the search key. Returned values If successful, the function returns the ID of the `KosString`, otherwise it returns `RTL_NULL`.
`KosCreateString()`	Purpose Adds a `KosString` to the default hash table. Parameters • [in] `str` – pointer to the C-string whose data needs to be put into the `KosString`.

	• [out] `outStr` – pointer to the ID of the `KosString`. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosCreateStringEx()`	Purpose Adds a `KosString` to a defined hash table. Parameters • [in,optional] `root` – ID of the hash table, or `RTL_NULL` to assign the default hash table. • [in] `str` – pointer to the C-string whose data needs to be put into the `KosString`. • [out] `outStr` – pointer to the ID of the `KosString`. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KosRefString()`	Purpose Increments the counter for links to a `KosString`. Parameters • [in] `str` – ID of the `KosString`. Returned values N/A
`KosRefStringEx()`	Purpose Increments the counter for links to a `KosString`. Parameters • [in,optional] `root` – fictitious parameter. You must specify `RTL_NULL`. • [in] `str` – ID of the `KosString`. Returned values N/A
`KosPutString()`	Purpose Decrements the counter for links to a `KosString` in the default hash table. Parameters • [in] `str` – ID of the `KosString`. Returned values N/A
`KosPutStringEx()`	Purpose Decrements the counter for links to a `KosString` in a defined hash table. Parameters • [in,optional] `root` – ID of the hash table, or `RTL_NULL` to assign the default hash table. • [in] `str` – ID of the `KosString`. Returned values N/A

`KosGetStringLen()`	Purpose Gets the size of the `KosString` without taking into account the terminating zero. Parameters • [in] `str` – ID of the `KosString`. Returned values Size of the `KosString` without taking into account the terminating zero, in bytes. Additional information If the parameter has the `RTL_NULL` value, it returns `0`.
`KosGetStringSize()`	Purpose Gets the size of the `KosString` while taking into account the terminating zero. Parameters • [in] `str` – ID of the `KosString`. Returned values Size of the `KosString` while taking into account the terminating zero, in bytes. Additional information If the parameter has the `RTL_NULL` value, it returns `0`.

KasperskyOS kernel XHCI DbC driver management (xhcidbg_api.h)

This API is defined in the header file `sysroot-*-kos/include/coresrv/xhcidbg/xhcidbg_api.h` from the KasperskyOS SDK.

This API is intended for use in the code of drivers that implement a USB stack (kusb class drivers) and lets you stop and start the XHCI DbC (Debug Capability) driver of the KasperskyOS kernel. (The KasperskyOS kernel includes the XHCI DbC driver, which is a simplified XHCI controller driver and is used by the kernel for diagnostic output or by the kernel GDB server to interact with the GDB debugger.) If a kusb class driver needs to restart the XHCI controller, the kernel's XHCI DbC driver must be stopped by calling the `KnXhcidbgStop()` function before the restart and then started by calling the `KnXhcidbgStart()` function after the restart. If you do not do this, a restart of the XHCI controller will lead to a failure of diagnostic output (or debugging) via the USB port.

> This API has not been implemented for the Raspberry Pi 4 B and Radxa ROCK 3A hardware platforms (the functions return `rcUnimplemented`).

Information about API functions is provided in the table below.

xhcidbg_api.h functions

Function	Information about the function
`KnXhcidbgStart()`	Purpose Starts the XHCI DbC driver of the KasperskyOS kernel. Parameters

	N/A Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnXhcidbgStop()`	Purpose Stops the XHCI DbC driver of the KasperskyOS kernel. Parameters N/A Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Receiving security audit data (vlog_api.h)

This API is defined in the header file `sysroot-*-kos/include/coresrv/vlog/vlog_api.h` from the KasperskyOS SDK.

The API lets you read from the KasperskyOS kernel log containing security audit data and is used in the code of the static library `sysroot-*-kos/lib/libklog_system_audit.a`, which is linked to the `Klog` system program.

Information about API functions is provided in the table below.

Using the API

The API usage scenario includes the following steps:

1. Open the kernel log containing security audit data by calling the `KnAuOpen()` function.
2. Receive messages from the kernel log containing security audit data via calls of the `KnAuRead()` function.

   This log is a cyclic buffer, and therefore you must prevent overwrites in this log because they could lead to a loss of security audit data that has not yet been read. To monitor overwrites in the log, the `outDropMsgs` parameter of the `KnAuRead()` function is used to pass the number of dropped messages. (The counter of these messages is reset to zero after this value is read each time the function is called.) If messages have been dropped, you must either increase the speed at which messages are read from the log, for example, by performing the read operation from parallel threads, or reduce the speed of message generation by editing the security audit profile.
3. Close the kernel log containing security audit data by calling the `KnAuClose()` function.

Information about API functions

vlog_api.h functions

Function	Information about the function
`KnAuOpen()`	Purpose Opens the kernel log containing security audit data. Parameters • [in] `name` – pointer to the name of the kernel log containing security audit data (`kss` must be specified).

	• [out] `outRID` – pointer to the handle of the kernel log containing security audit data. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnAuClose()`	Purpose Closes the kernel log containing security audit data. Parameters • [in] `rid` – handle of the kernel log containing security audit data. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnAuRead()`	Purpose Receives a message from the kernel log containing security audit data. Parameters • [in] `rid` – handle of the kernel log containing security audit data. • [out] `msg` – pointer to the buffer used to store the message. The type of message is defined in the header file `sysroot-*-kos/include/vlog/audit.h`. The buffer size must be at least `FULL_AUDIT_MESSAGE_LENGTH_MAX` bytes. • [out] `outDropMsgs` – pointer to the number of messages that were lost due to overwrites in the kernel log containing security audit data. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Using futexes (sync.h)

This API is defined in the header file `sysroot-*-kos/include/coresrv/sync/sync.h` from the KasperskyOS SDK.

The API is intended for working with futexes and is used in code of the API functions in event.h, mutex.h, rwlock.h, semaphore.h, and condvar.h. A *futex* is a low-level synchronization primitive that supports two operations: lock a thread and add it to the futex-linked queue of locked threads, and resume execution of threads from the futex-linked queue of locked threads. A futex is a kernel object linked to an integer variable in the user space. The kernel object provides the capability to store a queue of locked threads linked to the futex. The value of the integer variable in the user space (futex value) is atomically modified by the synchronized threads to signal the changed state of shared resources. For example, the futex value may indicate the specific state of an event (signaling or non-signaling state), indicate whether the mutex has been captured or freed, and indicate the specific value of the semaphore counter. The futex value determines whether to lock a thread that attempts to obtain access to shared resources.

Information about API functions is provided in the table below.

Using the API

To use a futex, you must create only an integer variable for storing its value. The API functions receive the pointer to this variable via the `ftx` parameter. The kernel object linked to this variable is created or deleted automatically when using the API functions.

The `KnFutexWait()` function locks the calling thread if the futex value matches the value of the `val` parameter. (A futex value can be changed by another thread during execution of a function.) The thread is locked for the `mdelay` time period, but execution of this thread may be resumed before the `mdelay` period elapses by calling the `KnFutexWake()` function from another thread. The `KnFutexWake()` function resumes execution of threads from a futex-linked queue of locked threads. The number of threads whose execution is resumed is limited by the value of the `nThreads` parameter. Thread execution is resumed starting with the beginning of the queue.

Information about API functions

sync.h functions

Function	Information about the function
`KnFutexWait()`	Purpose Locks the calling thread if the futex value is equal to the expected value. Parameters • [in] `ftx` – pointer to the variable containing the futex value. • [in] `val` – expected value of the futex. • [in] `mdelay` – maximum lockout duration in milliseconds, or `INFINITE_TIMEOUT` to set an unlimited lockout duration. • [out,optional] `outDelay` – actual lockout duration in milliseconds, or `RTL_NULL` if this information is not required. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If the thread lockout duration has expired, the function returns `rcTimeout`. If the futex value is not equal to the expected value, the function returns `rcFutexWouldBlock`.
`KnFutexWake()`	Purpose Resumes execution of threads that were blocked by `KnFutexWait()` function calls with the defined futex. Parameters • [in] `ftx` – pointer to the variable containing the futex value. • [in] `nThreads` – maximum number of threads whose execution can be resumed. • [out,optional] `wokenCnt` – actual number of threads whose execution has been resumed, or `RTL_NULL` if this information is not required. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Getting IPC handles and RIIDs to use statically created IPC channels (sl_api.h)

This API is defined in the header file `sysroot-*-kos/include/coresrv/sl/sl_api.h` from the KasperskyOS SDK.

The API allows servers to get the listener handles, and allows clients to get the client IPC handles and Runtime Implementation Identifiers (RIIDs) for the purpose of using statically created IPC channels.

> Closing an obtained IPC handle will cause the IPC channel to become unavailable. After an IPC handle is closed, it is impossible to obtain it again or restore access to the IPC channel.

Information about API functions is provided in the table below.

sl_api.h functions

Function	Information about the function
`ServiceLocatorRegister()`	Purpose Gets the listener handle based on the name of the IPC channel. Parameters • [in] `channelName` – pointer to the name of the IPC channel. • [in] `endpoints` – fictitious parameter (obsolete). It must be set to `RTL_NULL`. • [in] `endpointsSize` – fictitious parameter (obsolete). It must be set to `0`. • [out] `id` – fictitious parameter (obsolete). Returned values If successful, it returns the listener handle. Otherwise, it returns `INVALID_HANDLE`.
`ServiceLocatorConnect()`	Purpose Gets the client IPC handle based on the name of the IPC channel. Parameters • [in] `channelName` – pointer to the name of the IPC channel. Returned values If successful, it returns the client IPC handle. Otherwise, it returns `INVALID_HANDLE`.
`ServiceLocatorGetRiid()`	Purpose Gets the endpoint ID (RIID). Parameters • [in] `handle` – client IPC handle. • [in] `name` – pointer to the qualified name of the endpoint. For information about the qualified name of an endpoint, see Binding methods of security models to security events. Returned values If successful, the function returns the endpoint ID (RIID), otherwise it returns `INVALID_RIID`. The type of returned value is defined in the header file `sysroot-*-kos/include/nk/types.h` from the KasperskyOS SDK.

Power management (pm_api.h)

This API is defined in the header file `sysroot-*-kos/include/coresrv/pm/pm_api.h` from the KasperskyOS SDK.

The API changes the power mode of the hardware platform (for example, shutdown or restart), and enables and disables processors (processor cores).

Information about API functions is provided in the table below.

Using the API

To make a request to change the power mode of the hardware platform, call the `KnPmRequest()` function.

The `KnPmSetCpusOnline()` and `KnPmGetCpusOnline()` functions have the `mask` parameter, which is used to pass the processor mask. A *processor mask* is a bit mask indicating the set of processors (processor cores). A flag set in the *i*th bit indicates that a processor with the *i* index is included in the set (numbering starts at zero). To work with a processor mask, you must use the functions defined in the header file `sysroot-*-kos/include/rtl/cpuset.h`.

To enable and/or disable processors, call the `KnPmSetCpusOnline()` function while specifying the processor mask indicating the required set of active processors. The flag corresponding to the bootstrap processor must be set in this mask, otherwise the function will end with an error. To get the bootstrap processor index, call the `KnPmGetBootstrapCpuNum()` function.

To find out which processors are in the active state, call the `KnPmGetCpusOnline()` function. The output parameter of this function is the processor mask indicating the actual set of active processors.

If the `KnPmSetCpusOnline()` function ended with an error, this could lead to an unpredictable change in the set of active processors. To make a repeated attempt to configure the set of active processors, you must find out which processors are in the active state by calling the `KnPmGetCpusOnline()` function and use this information to adjust the required set of active processors during the subsequent `KnPmSetCpusOnline()` function call.

Information about API functions

pm_api.h functions

Function	Information about the function
`KnPmRequest()`	Purpose Requests to change the power mode of a hardware platform. Parameters • [in] `request` – value defining the necessary power mode of the hardware platform. The parameter type is defined in the header file `sysroot-*-kos/include/pm/pmstate.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnPmSetCpusOnline()`	Purpose Requests to enable and/or disable processors. Parameters • [in] `mask` – pointer to the processor mask indicating the required set of active processors. The type of processor mask is defined in the header file `sysroot-*-kos/include/rtl/cpuset.h` from the KasperskyOS SDK. • [in] `timeoutMs` – fictitious parameter (obsolete). Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnPmGetCpusOnline()`	Purpose Gets information regarding which processors are in the active state.

	Parameters • [out] `mask` – pointer to the processor mask indicating the actual set of active processors. The type of processor mask is defined in the header file `sysroot-*-kos/include/rtl/cpuset.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`KnPmGetBootstrapCpuNum()`	Purpose Gets the index of the bootstrap processor. Parameters • [out] `num` – pointer to the bootstrap processor index. The type of processor index is defined in the header file `sysroot-*-kos/include/rtl/cpuset.h` from the KasperskyOS SDK. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

POSIX support

POSIX support limitations

KasperskyOS has a limited implementation of POSIX oriented toward the POSIX.1-2008 standard. These limitations are primarily due to security precautions.

> There is no XSI support or optional functionality.

Limitations affect the following:

- Interaction between processes
- Interaction between threads via signals
- Asynchronous input/output
- Use of robust mutexes
- Terminal operations
- Shell operations
- File handle management
- Clock usage
- Getting system parameters

Limitations include:

- Unimplemented interfaces
- Interfaces that are implemented with deviations from the POSIX.1-2008 standard
- Stub interfaces that do not perform any operations except assign the `ENOSYS` value to the `errno` variable and return the value `-1`

> In KasperskyOS, signals cannot interrupt the `Call()`, `Recv()`, and `Reply()` system calls that support the operation of libraries that implement the POSIX interface.
> The KasperskyOS kernel does not transmit signals.
> On hardware platforms running an Arm processor architecture, input and output parameters of POSIX interfaces cannot be saved in "Device memory" because this could lead to unpredictable behavior. Parameters of POSIX interfaces should be saved in "Normal memory". To copy data from "Device memory" to "Normal memory" and vice versa, you must use the `RtlPedanticMemcpy()` function declared in the header file `sysroot-*-kos/include/rtl/string_pedantic.h` from the KasperskyOS SDK.

Limitations on interaction between processes

Interface	Purpose	Implementation	Header file based on the POSIX.1-2008 standard
`fork()`	Create a new (child) process.	Not implemented	`unistd.h`
`pthread_atfork()`	Register the handlers that are called before and after the child process is created.	Not implemented	`pthread.h`
`wait()`	Wait for the child process to stop or complete.	Stub	`sys/wait.h`
`waitid()`	Wait for the state of the child process to change.	Not implemented	`sys/wait.h`
`waitpid()`	Wait for the child process to stop or complete.	Stub	`sys/wait.h`
`execl()`	Run the executable file.	Not implemented	`unistd.h`
`execle()`	Run the executable file.	Not implemented	`unistd.h`
`execlp()`	Run the executable file.	Not implemented	`unistd.h`
`execv()`	Run the executable file.	Not implemented	`unistd.h`
`execve()`	Run the executable	Not implemented	`unistd.h`

	file.		
`execvp()`	Run the executable file.	Not implemented	`unistd.h`
`fexecve()`	Run the executable file.	Not implemented	`unistd.h`
`setpgid()`	Move the process to another group or create a group.	Stub	`unistd.h`
`setsid()`	Create a session.	Stub	`unistd.h`
`getpgrp()`	Get the group ID of the calling process.	Stub	`unistd.h`
`getpgid()`	Get the group ID.	Stub	`unistd.h`
`getppid()`	Get the ID of the parent process.	Stub	`unistd.h`
`getsid()`	Get the session ID.	Stub	`unistd.h`
`times()`	Get the time values for the process and its descendants.	Stub	`sys/times.h`
`kill()`	Send a signal to the process or group of processes.	Only the `SIGTERM` signal can be sent. The `pid` parameter is ignored.	`signal.h`
`pause()`	Wait for a signal.	Stub	`unistd.h`
`sigpending()`	Check for received blocked signals.	Not implemented	`signal.h`
`sigqueue()`	Send a signal to the process.	Not implemented	`signal.h`
`sigtimedwait()`	Wait for a signal from the defined set of signals.	Not implemented	`signal.h`
`sigwaitinfo()`	Wait for a signal from the defined set of signals.	Not implemented	`signal.h`
`sem_init()`	Create an unnamed semaphore.	You cannot create an unnamed semaphore for synchronization between processes. If a non-zero value is passed through the `pshared` parameter, it will return only the value `-1` and will assign the `ENOTSUP` value to the `errno` variable.	`semaphore.h`
`sem_open()`	Create/open a named semaphore.	You cannot open a named semaphore that was created by another process. Named semaphores (like unnamed semaphores) are local, which means that they are accessible only to the process that created them.	`semaphore.h`
`pthread_spin_init()`	Create a spin lock.	You cannot create a spin lock for synchronization between processes. If the `PTHREAD_PROCESS_SHARED` value is passed through the `pshared` parameter, this value will be ignored.	`pthread.h`

mmap()	Map to memory.	You cannot perform memory mapping for interaction between processes. If the MAP_SHARED and PROT_WRITE values are passed through the flags and prot parameters, respectively, it will return only the MAP_FAILED value and will assign the EACCES value to the errno variable. For all other possible values of the prot parameter, the MAP_SHARED value of the flags parameter is ignored. In addition, you cannot use the prot parameter to pass combinations of the PROT_WRITE|PROT_EXEC and PROT_READ|PROT_WRITE|PROT_EXEC flags, or combinations with the PROT_READ|PROT_EXEC flag under certain configurations of the KasperskyOS kernel. In this case, it will return only the MAP_FAILED value and will assign the ENOMEM value to the errno variable.	sys/mman.h
mprotect()	Define the memory access permissions.	For security purposes, some configurations of the KasperskyOS kernel prohibit granting simultaneous write-and-execute access to virtual memory regions. If this type of kernel configuration is in use and you pass the PROT_WRITE|PROT_EXEC value through the prot parameter, it will only return the -1 value and will assign the ENOTSUP value to the errno variable.	sys/mman.h
pipe()	Create an unnamed channel.	You cannot use an unnamed channel for data transfer between processes. Unnamed channels are local, which means that they are accessible only to the process that created them.	unistd.h
mkfifo()	Create a special FIFO file (named channel).	Stub	sys/stat.h
mkfifoat()	Create a special FIFO file (named channel).	Not implemented	sys/stat.h

Limitations on interaction between threads via signals

Interface	Purpose	Implementation	Header file based on the POSIX.1-2008 standard
pthread_kill()	Send a signal to a thread.	You cannot send a signal to a thread. If a signal number is passed through the sig parameter, only the ENOSYS value is returned.	signal.h
siglongjmp()	Restore the state of the control thread and the signals mask.	Not implemented	setjmp.h
sigsetjmp()	Save the state of the control thread and the signals mask.	Not implemented	setjmp.h

Asynchronous input/output limitations

Interface	Purpose	Implementation	Header file based on the POSIX.1-2008 standard
aio_cancel()	Cancel input/output requests that are waiting to be handled.	Not implemented	aio.h
aio_error()	Receive an error from an asynchronous input/output operation.	Not implemented	aio.h
aio_fsync()	Request the execution of input/output operations.	Not implemented	aio.h
aio_read()	Request a file read operation.	Not implemented	aio.h
aio_return()	Get the status of an asynchronous input/output operation.	Not implemented	aio.h
aio_suspend()	Wait for the completion of asynchronous input/output operations.	Not implemented	aio.h
aio_write()	Request a file write operation.	Not implemented	aio.h

`lio_listio()`	Request execution of a set of input/output operations.	Not implemented	`aio.h`

Limitations on the use of robust mutexes

Interface	Purpose	Implementation	Header file based on the POSIX.1-2008 standard
`pthread_mutex_consistent()`	Return a robust mutex to a consistent state.	Not implemented	`pthread.h`
`pthread_mutexattr_getrobust()`	Get a robust mutex attribute.	Not implemented	`pthread.h`
`pthread_mutexattr_setrobust()`	Define a robust mutex attribute.	Not implemented	`pthread.h`

Terminal operation limitations

Interface	Purpose	Implementation	Header file based on the POSIX.1-2008 standard
`ctermid()`	Get the path to the file of the control terminal.	Only returns or passes an empty string through the `s` parameter.	`stdio.h`
`tcsetattr()`	Define the terminal settings.	The input speed, output speed, and other settings specific to hardware terminals are ignored.	`termios.h`
`tcdrain()`	Wait for output completion.	Returns only the value `-1`.	`termios.h`
`tcflow()`	Suspend or resume receipt or transmission of data.	Suspending output and resuming suspended output are not supported.	`termios.h`
`tcflush()`	Clear the input queue or output queue, or both of these queues.	Returns only the value `-1`.	`termios.h`
`tcsendbreak()`	Break the connection with the terminal for a set time.	Returns only the value `-1`.	`termios.h`
`ttyname()`	Get the path to the terminal file.	Returns only a null pointer.	`unistd.h`
`ttyname_r()`	Get the path to the terminal file.	Returns only an error value.	`unistd.h`
`tcgetpgrp()`	Get the ID of a group of processes using the terminal.	Returns only the value `-1`.	`unistd.h`
`tcsetpgrp()`	Define the ID for a group of processes using the terminal.	Returns only the value `-1`.	`unistd.h`
`tcgetsid()`	Get the ID of a group of processes for the leader of the session connected to the terminal.	Returns only the value `-1`.	`termios.h`

Shell operation limitations

Interface	Purpose	Implementation	Header file based on the POSIX.1-2008 standard
`popen()`	Create a child process for command execution and an unnamed channel with this process.	Not implemented	`stdio.h`
`pclose()`	Close the unnamed channel with the child process created by `popen()`, and wait for the child process to terminate.	Not implemented	`stdio.h`
`system()`	Create a child process for command execution.	Stub	`stdlib.h`

`wordexp()`	Perform a shell-like expansion of the string.	Not implemented	`wordexp.h`
`wordfree()`	Free up the memory allocated for the results from calling `wordexp()`.	Not implemented	`wordexp.h`

Limitations on file handle management

Interface	Purpose	Implementation	Header file based on the POSIX.1-2008 standard
`dup()`	Make a copy of the handle of an opened file.	Handles of regular files, standard I/O streams, sockets and channels are supported. There is no guarantee that the lowest available handle will be received.	`fcntl.h`
`dup2()`	Make a copy of the handle of an opened file.	Handles of regular files, standard I/O streams, sockets and channels are supported. The handle of an opened file needs to be passed through the `fildes2` parameter.	`fcntl.h`

Limitations on clock usage

Interface	Purpose	Implementation	Header file based on the POSIX.1-2008 standard
`clock_gettime()`	Get the time value.	If the `CLOCK_PROCESS_CPUTIME_ID` value or `CLOCK_THREAD_CPUTIME_ID` value is passed through the `clock_id` parameter, it will return only the value `-1` and will assign the `EINVAL` value to the `errno` variable.	`time.h`
`clock()`	Get the CPU time spent on execution of the calling process.	Returns the amount of time (in milliseconds) that has elapsed since the KasperskyOS kernel was started.	`time.h`

Getting system parameters

Interface	Purpose	Implementation	Header file based on the POSIX.1-2008 standard
`confstr()`	Get a system parameter.	Stub	`unistd.h`

POSIX implementation specifics

In KasperskyOS, the specific implementation of some POSIX interfaces not entirely defined by the POSIX.1-2008 standard differs from the implementation of these interfaces in Linux and other UNIX-like operating systems. Information about these interfaces is provided in the table below.

POSIX interfaces and their implementation specifics

Interface	Purpose	Implementation	Header file based on the POSIX.1-2008 standard
`bind()`	Assign a name to a socket.	When using a VFS version that supports only network operations, files of sockets in the `AF_UNIX` family are saved in a special file system implemented by this VFS version when `bind()` is called. A socket file can be created only in the root of the file system or in the `/tmp` directory, and it can be re-used after the socket is closed.	`sys/socket.h`
`mmap()`	Map to memory.	The `MAP_FIXED` flag in the `flags` parameter is not supported.	`sys/mman.h`

		Mapping more than 4 GB is not supported on hardware platforms running an AArch64 (Arm64) processor architecture.	
`read()`	Read from a file.	If the size of the `buf` buffer exceeds the size of the read data, the remainder of this buffer is filled with zeros.	`unistd.h`

Concurrently using POSIX and the libkos API

In a thread that is created using Pthreads, you cannot use the following `libkos` APIs:

- event.h, mutex.h, rwlock.h, semaphore.h, condvar.h
- thread.h, thread_api.h
- dma.h
- ports.h
- mmio.h
- irq.h

The following `libkos` APIs can be used together with Pthreads (and other POSIX APIs):

- handle_api.h
- notice_api.h
- task.h, task_api.h
- cm_api.h, ns_api.h
- queue.h

POSIX interfaces cannot be used in threads that were created using the thread.h and thread_api.h APIs.

The syscalls.h API can be used in any threads that were created using Pthreads or the thread.h and thread_api.h APIs.

Obtaining statistical data on the system through the libc library API

The `libkos` library provides APIs that let you obtain statistical data on file systems and network interfaces managed by VFS. The functions of these APIs are presented in the table below.

Information on file systems and network interfaces

Function	Header file from the KasperskyOS SDK	Obtained information
`statvfs()`	`sysroot-*-kos/include/strict/posix/sys/statvfs.h`	File system information, such as the block size, number of blocks, and number of available blocks
`getvfsstat()`	`sysroot-*-kos/include/sys/statvfs.h`	The information on all mounted file systems is identical to the information provided by the `statvfs()` function
`getifaddrs()`	`sysroot-*-kos/include/ifaddrs.h`	Information on network interfaces, such as their name, IP address, and subnet mask

Dynamically creating IPC channels using the DCM system program

The KasperskyOS SDK provides the `DCM` (Dynamic Connection Manager) system program, which lets you dynamically create IPC channels. This program enables servers to notify clients about their provided endpoints, and pass callable handles for using these endpoints.

To use the `DCM` program, the KasperskyOS SDK provides the following files:

- `sysroot-*-kos/include/dcm/dcm_api.h` – header file that defines the basic API
- `sysroot-*-kos/include/dcm/groups/pub.h` – header file that defines the additional API for servers
- `sysroot-*-kos/include/dcm/groups/subscr.h` – header file that defines the additional API for clients
- `sysroot-*-kos/include/dcm/dcm.h` – header file for including clients and servers into source code
- `sysroot-*-kos/bin/dcm` – executable file
- `sysroot-*-kos/lib/libdcm.a` – client library
- `sysroot-*-kos/lib/libdcm_s.a` – client library built with a `-fPIC` flag to ensure the possibility to link to a dynamic library
- `sysroot-*-kos/include/kl/core/DCM.*dl` – formal specification files

IPC channels from clients and servers to the `DCM` must be statically created.

You can run multiple `DCM` processes. Each client and server can be connected to only one `DCM` process.

The `DCM` API uses IDs known as *DCM handles* . DCM handles identify objects that provide the following capabilities to clients and servers that use `DCM`:

- Clients can receive notifications about the publishing and unpublishing of endpoints, and can make requests to create IPC channels to servers.
- Servers can receive client requests to create IPC channels, and can accept or decline these requests.

The names of endpoints and endpoints interfaces should be defined according to the formal specifications of solution components. (For information about the qualified name of an endpoint, see Binding methods of security models to security events.) Instead of the qualified name of an endpoint, you can use any conditional name of this endpoint. The names of clients and servers are defined in the init description.

Using the base API (dcm_api.h)

Information about API functions is provided in the table below.

Dynamically creating an IPC channel on the server side

Dynamic creation of an IPC channel on the server side includes the following steps:

1. Create a connect to `DCM` by calling the `DcmInit()` function.

2. Publish the provided endpoints to `DCM` by using the `DcmPublishEndpoint()` function.

   In the `serverHandle` parameter of the `DcmPublishEndpoint()` function, specify the handle that identifies the server. The server-identifying handle can be the handle of this server process created by calling of the `KosTaskGetSelf()` and `KosTaskGetHandle()` functions from the task.h API in a sequence, or the handle created by calling the `KnHandleCreateUserObject()` function from the handle_api.h API. (When calling the `KnHandleCreateUserObject()` function, the `rights` parameter must specify the `OCAP_HANDLE_TRANSFER` and `OCAP_HANDLE_GET_SID` flags.) You can publish multiple endpoints by specifying the same or different handles that identify the server. (Clients can receive the descendants of these handles via the optional `outServerHandle` parameter of the `DcmReadPubQueue()` function.) After the endpoints are published, the handle identifying the server can be closed.

   A `DcmPublishEndpoint()` function call creates a DCM handle that enables receipt of client requests to create IPC channels to use a specific endpoint with a specific interface.

   To unpublish an endpoint, call the `DcmCloseHandle()` function to close the DCM handle that was created when the endpoint was published. If a server terminates, all of its published endpoints will also be unpublished.

3. Receive a client request to create an IPC channel by calling the `DcmListen()` function.

   By calling the `DcmListen()` function with the same DCM handle in the `pubHandle` parameter, you can receive just those client requests that were sent to create IPC channels to use the endpoint with the interface that was defined when creating this DCM handle at step 2.

   You can use the optional `outClientHandle` and `outClientName` parameters of the `DcmListen()` function to get the handle identifying the client and the client name, respectively. (The handle identifying the client is the descendant of the handle specified in the `clientHandle` parameter of the `DcmCreateConnection()` function called on the client side.) This data can be used to make a decision on whether to accept or reject a client request. For example, the handle identifying the client may be used to get information about the client from other processes or to get the SID for this handle to compare it to other client-identifying handles available to the server. After making a decision, you can close the handle that identifies the client.

   A `DcmListen()` function call creates a DCM handle that enables acceptance or rejection of a client request to create an IPC channel.

4. You can accept a client request to create an IPC channel by calling the `DcmAccept()` function.

   In the `connectionId` parameter of the `DcmAccept()` function, specify the DCM handle that was obtained at step 3.

   In the `sessionHandle` parameter of the `DcmAccept()` function, specify the callable handle to be passed to the client. To create a callable handle, call the `KnHandleCreateUserObjectEx()` function from the handle_api.h API. (When calling the `KnHandleCreateUserObjectEx()` function, the `rights` parameter must specify the `OCAP_IPC_CALL` and `OCAP_HANDLE_TRANSFER` flags, and the `ipcChannel` and `riid` parameters must specify the server IPC handle of the created IPC channel and the RIID, respectively. The server IPC handle is used to initialize IPC transport and manage IPC request processing on the server side. You can use the `KnHandleCreateListener()` function from the handle_api.h API to create a server handle. The endpoint ID (RIID) is a constant in the automatically generated transport code, for example: `FsDriver_operationsComp_fileOperations_iid`. After the client request is accepted, the callable handle can be closed.

   The transfer of a callable handle to a client can be associated with a resource transfer context object by specifying the handle of this object in the `badgeHandle` parameter of the `DcmAccept()` function. This will let you use the notice_api.h API to track the closing or revocation of descendants of the callable handle that form the handle inheritance subtree whose root node was generated when this callable handle was transferred to the client. (The transferred callable handle can be closed by the client, but this handle is also closed when the client terminates.) To create a resource transfer context object, call the `KnHandleCreateBadge()` function from the handle_api.h API.

You can use the `context` parameter of the `KnHandleCreateUserObjectEx()` and `KnHandleCreateBadge()` functions to define the data to be associated with the callable handle and the transfer of the callable handle, respectively (similar to the user resource context and resource transfer context). The server will be able to receive the pointer to this data when dereferencing the callable handle. (Even though the callable handle is an IPC handle, the kernel puts it into the `base_.self` field of the constant part of an IPC request.)

The same callable handle can be used to accept multiple requests from one or more clients.

To reject a client request to create an IPC channel, call the `DcmCloseHandle()` function to close the DCM handle obtained at step 3. (In this case, a `DcmConnect()` function call on the client side terminates with an `rcNotConnected` error.) If a client request is accepted, the DCM handle obtained at step 3 must also be closed but only after calling the `DcmAccept()` function.

Dynamically creating an IPC channel on the client side

Dynamic creation of an IPC channel on the client side includes the following steps:

1. Create a connect to `DCM` by calling the `DcmInit()` function.

2. Create a DCM handle that enables receipt of a notification about publishing and unpublishing of endpoints with a specific interface.

   To complete this step, call the `DcmSetSubscription()` function.

   Use the optional parameters `endpointName`, `serverName` and `serverHandle` of the `DcmSetSubscription()` function to include notification filtering by qualified endpoint name, server name, and server-identifying handle, respectively. (The handle identifying the server is the descendant of the handle specified in the `serverHandle` parameter of the `DcmPublishEndpoint()` function called on the server side. One way for the client to get this handle is from another process.)

   The `DcmSetSubscription()` function call creates a DCM handle that enables receipt of notifications about the publishing and unpublishing of endpoints with a specific interface. It also generates a queue of notifications about the publication of endpoints that were published prior to the `DcmSetSubscription()` call and match the filter criteria. If you no longer need to receive notifications about publishing and unpublishing of endpoints with a specific interface, close this DCM handle by calling the `DcmCloseHandle()` function.

3. You can extract a notification about endpoint publishing from the queue of these notifications by calling the `DcmReadPubQueue()` function.

   In the `subscrHandle` parameter of the `DcmReadPubQueue()` function, specify the DCM handle that was obtained at step 2.

   You can use the `outServerHandle`, `outServerName` and `outEndpointName` parameters of the `DcmReadPubQueue()` function to get the handle identifying the server, the server name, and the qualified name of the endpoint, respectively. (The handle identifying the server is the descendant of the handle specified in the `serverHandle` parameter of the `DcmPublishEndpoint()` function called on the server side.) This data can be used to decide whether to fulfill a request to create an IPC channel to the server. For example, the handle identifying the server may be used to get information about the server from other processes or to get the SID for this handle to compare it to other server-identifying handles available to the client.

   The value received via the `outPubStatus` parameter of the `DcmReadPubQueue()` function indicates that the endpoint was published or unpublished. A notification about endpoint unpublishing appears if the server unpublished this endpoint, or if the server terminated.

4. Create a DCM handle that enables fulfillment of a request to create an IPC channel to the server.

   To complete this step, call the `DcmCreateConnection()` function.

   In the `serverHandle` parameter of the `DcmCreateConnection()` function, specify the handle that identifies the server. This handle can be obtained at step 3 via the `outServerHandle` parameter of the `DcmReadPubQueue()` function or by other means, such as from another process.

In the `clientHandle` parameter of the `DcmCreateConnection()` function, specify the handle that identifies the client. This client-identifying handle can be the handle of the client process created by calling of the `KosTaskGetSelf()` and `KosTaskGetHandle()` functions from the task.h API in a sequence, or the handle created by calling the `KnHandleCreateUserObject()` function from the handle_api.h API. (When calling the `KnHandleCreateUserObject()` function, the `rights` parameter must specify the `OCAP_HANDLE_TRANSFER` and `OCAP_HANDLE_GET_SID` flags.) You can create multiple DCM handles that enable fulfillment of a request to create an IPC channel by specifying the same or different handles identifying the client in the `clientHandle` parameter of the `DcmCreateConnection()` function. (Servers can receive the descendants of these handles via the optional `outClientHandle` parameter of the `DcmListen()` function.) After creating a DCM handle that enables fulfillment of a request to create an IPC channel, the handle identifying the client can be closed.

A `DcmCreateConnection()` function call creates a DCM handle that enables completion of a request to create an IPC channel to the server to use a specific endpoint with a specific interface.

5. Complete a request to create an IPC channel to the server by calling the `DcmConnect()` function.

   In the `connectionId` parameter, specify the DCM handle obtained at step 4.

   After the `DcmConnect()` function is called, the client receives a callable handle via the `outSessionHandle` parameter. The client uses this handle to initialize IPC transport. In addition, the client specifies the `INVALID_RIID` value as the Runtime Implementation Identifier (RIID) in the proxy object initialization function.

   After receiving a callable handle, the DCM handle obtained at step 4 must be closed by calling the `DcmCloseHandle()` function.

Interrupting blocking function calls

To interrupt a blocking call of the `DcmConnect()`, `DcmReadPubQueue()` or `DcmListen()` function from another thread, call the `DcmInterruptBlockingCall()` function.

Using notifications

You can use the `DcmSubscribeToEvents()` and `DcmUnsubscribeFromEvents()` functions together with functions from the notice_api.h API to receive notifications about the occurrence of the following events: the server published or depublished an endpoint in `DCM` (the `DCM_PUBLICATION_CHANGED` flag), the server received a request from the client to create an IPC channel (the `DCM_CLIENT_CONNECTED` flag), a blocking call of the `DcmConnect()`, `DcmReadPubQueue()` or `DcmListen()` function was interrupted (the `DCM_BLOCKING_CALL_INTERRUPTED` flag), or the server received or rejected a client request to create an IPC channel (the `DCM_CLIENT_RELEASED_BY_SERVER` flag). (Flags of the event mask and the "resource–event mask" entry ID `DCM_EVENT_ID` are defined in the header file `sysroot-*-kos/include/dcm/dcm_api.h` from the KasperskyOS SDK.) The `DcmSubscribeToEvents()` and `DcmUnsubscribeFromEvents()` functions let you configure the notification receiver by adding or deleting, respectively, tracked objects identified by DCM handles.

Deleting dynamically created IPC channels

A dynamically created IPC channel will be deleted when closing the callable handle and server IPC handle of this IPC channel.

Deleting a DCM connection

If you no longer need to use `DCM`, you need to call the `DcmFini()` function to delete the connection to `DCM` and thereby free up the resources linked to this connection.

Information about API functions

dcm_api.h functions

Function	Information about the function
`DcmInit()`	Purpose Creates a connection to `DCM`. Parameters N/A Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If there is no IPC channel to `DCM`, the function returns `rcNoCapability`. Additional information Thread-safe function. Non-blocking call.
`DcmFini()`	Purpose Deletes a connection to `DCM`. Parameters N/A Returned values N/A Additional information Thread-safe function. Non-blocking call.
`DcmSubscribeToEvents()`	Purpose Configures the notification receiver to receive notifications about events related to the object identified by the defined DCM handle. Parameters • [in] `notice` – identifier of the notification receiver. • [in] `handle` – DCM handle that was received by calling the `DcmSetSubscription()`, `DcmCreateConnection()` or `DcmPublishEndpoint()` function. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`DcmUnsubscribeFromEvents()`	Purpose Configures the notification receiver to not receive notifications about events related to the object identified by the defined DCM handle. Parameters • [in] `notice` – identifier of the notification receiver. • [in] `handle` – DCM handle that was received by calling the `DcmSetSubscription()`, `DcmCreateConnection()` or `DcmPublishEndpoint()` function. Returned values

	If successful, the function returns `rcOk`, otherwise it returns an error code.
`DcmCloseHandle()`	Purpose Closes the DCM handle obtained by calling the `DcmSetSubscription()`, `DcmCreateConnection()`, `DcmPublishEndpoint()` or `DcmListen()` function from the main API, and `DcmListenGroupPub()` from the additional API for servers. Parameters • [in] `handle` – DCM handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`DcmInterruptBlockingCall()`	Purpose Interrupts a blocking call of the `DcmConnect()`, `DcmReadPubQueue()` or `DcmListen()` function. Parameters • [in] `handle` – DCM handle specified when calling the `DcmConnect()`, `DcmReadPubQueue()` or `DcmListen()` function. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`DcmSetSubscription()`	Purpose Creates a DCM handle that enables receipt of a notification about publishing and unpublishing of endpoints with a specific interface. Parameters • [in] `endpointType` – pointer to the name of the interface of endpoints. • [in,optional] `endpointName` – pointer to the qualified name of the endpoint, or `RTL_NULL` to not use notification filtering based on the qualified name of the endpoint. • [in,optional] `serverName` – pointer to the server name, or `RTL_NULL` to not use notification filtering based on the server name. • [in,optional] `serverHandle` – handle identifying the server, or `INVALID_HANDLE` to not use notification filtering based on a specific server. • [out] `outSubscrHandle` – pointer to the DCM handle that enables receipt of notifications about the publishing and unpublishing of endpoints with a specific interface. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`DcmReadPubQueue()`	Purpose Extracts a notification about the publishing and unpublishing of an endpoint from the notification queue. Parameters • [in] `subscrHandle` – DCM handle created by calling the `DcmSetSubscription()` function. • [in] `msec` – time-out before notifications appearing in milliseconds, or `INFINITE_TIMEOUT` to set an unlimited time-out. • [out,optional] `outServerHandle` – pointer to the handle identifying the server, or `RTL_NULL` if this handle is not required. • [out,optional] `outServerName` – pointer to the buffer for the server name, or `RTL_NULL` if this name is not required. • [in] `serverNameSize` – size (bytes) for the server name, or `0` if this name is not required.

	• [out,optional] `outEndpointName` – pointer to the buffer for the qualified name of the endpoint, or `RTL_NULL` if this name is not required. • [in] `endpointNameSize` – size (bytes) of the buffer for the qualified name of the endpoint, or `0` if this name is not required. • [out] `outPubStatus` – pointer to the value indicating that the endpoint was published (`DcmEndpointPublished`) or unpublished (`DcmEndpointUnpublished`). Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If a blocking call was interrupted by calling the `DcmInterruptBlockingCall()` function, `rcAbort` is returned. Additional information Non-blocking call if the `msec` parameter is set to `0`.
`DcmCreateConnection()`	Purpose Creates a DCM handle that enables fulfillment of a request to create an IPC channel to the server to use a specific endpoint with a specific interface. Parameters • [in] `endpointType` – pointer to the name of the endpoint interface. • [in] `endpointName` – pointer to the qualified name of the endpoint. • [in] `serverHandle` – handle that identifies the server. • [in] `clientHandle` – handle that identifies the client. • [out] `outConnectionId` – pointer to the DCM handle that enables fulfillment of a request to create an IPC channel to the server to use a specific endpoint with a specific interface. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`DcmConnect()`	Purpose Fulfills a request to create an IPC channel with a server. Parameters • [in] `connectionId` – DCM handle created by calling the `DcmCreateConnection()` function. • [in] `msec` – timeout (in milliseconds) for fulfilling a request, or `INFINITE_TIMEOUT` to define an unlimited timeout. • [out] `outSessionHandle` – pointer to the callable handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If the `msec` parameter is set to `0` and the server did not handle the request to create an IPC channel, the function returns `rcNotReady`. If a blocking call was interrupted by calling the `DcmInterruptBlockingCall()` function, `rcAbort` is returned. Additional information Non-blocking call if the `msec` parameter is set to `0`.
`DcmPublishEndpoint()`	Purpose Publishes the endpoint to `DCM`. Parameters

	• [in] `endpointType` – pointer to the name of the endpoint interface. • [in] `endpointName` – pointer to the qualified name of the endpoint. • [in] `serverHandle` – handle that identifies the server. • [out] `outPubHandle` – pointer to the DCM handle that enables receipt of client requests to create IPC channels to use a specific endpoint with a specific interface. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`DcmListen()`	Purpose Receives a client request to create an IPC channel Parameters • [in] `pubHandle` – DCM handle created by calling the `DcmPublishEndpoint()` function. • [in] `msec` – timeout (in milliseconds) for the appearance of a client request, or `INFINITE_TIMEOUT` to define an unlimited timeout. • [out] `outConnectionId` – pointer to the DCM handle that enables you to accept or reject a client request to create an IPC channel. • [out,optional] `outClientHandle` – pointer to the handle identifying the client, or `RTL_NULL` if this handle is not required. • [out,optional] `outClientName` – pointer to the buffer for the client name, or `RTL_NULL` if this name is not required. • [in] `clientNameSize` – size (bytes) of the buffer for the client name, or `0` if this name is not required. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If a blocking call was interrupted by calling the `DcmInterruptBlockingCall()` function, `rcAbort` is returned. Additional information Non-blocking call if the `msec` parameter is set to `0`.
`DcmAccept()`	Purpose Accepts a client request to create an IPC channel. Parameters • [in] `connectionId` – DCM handle obtained by calling the `DcmListen()` function. • [in] `sessionHandle` – callable handle for the client. • [in,optional] `badgeHandle` – handle of the resource transfer context object, or `INVALID_HANDLE` if you do not need to associate the transfer of a callable handle with this object. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Using the additional server API for servers (pub.h)

The API allows servers to publish groups of endpoints. From the server perspective, a group of endpoints is a set of endpoints provided by the server and for which the server may receive client requests to create IPC channels using one DCM handle. A server can publish multiple groups of endpoints. A set of endpoints from different groups published by a server must not overlap.

Information about API functions is provided in the table below.

To publish a group of server-provided endpoints to `DCM`, call the `DcmPublishGroupEndpoints()` function. Use the `endpoints` parameter of the `DcmPublishGroupEndpoints()` function to define the qualified names and the contexts of grouped endpoints, and the names of endpoint interfaces. (The array of structures whose pointer is defined via the `endpoints` parameter can be deleted after publication.) An endpoint context is the data that is used by the server to determine which specific endpoint from the published group is required by the client when the `DcmListenGroupPub()` function is called. You can use a numerical identifier or a pointer to specific data as the endpoint context. A `DcmPublishGroupEndpoints()` function call creates a DCM handle that enables receipt of client requests to create IPC channels to use endpoints in the specific group with specific interfaces.

To receive a client request to create an IPC channel to use an endpoint in the published group, call the `DcmListenGroupPub()` function. In the `pubHandle` parameter of the `DcmListenGroupPub()` function, specify the DCM handle that was obtained by calling the `DcmPublishGroupEndpoints()` function. The output `outContext` of the `DcmListenGroupPub()` function contains the pointer to the endpoint context. The `DcmListenGroupPub()` function call creates a DCM handle that enables acceptance or rejection of a client request to create an IPC channel to use the specific endpoint in the group that matches the received endpoint context.

To unpublish a group of endpoints, call the `DcmCloseGroupPubHandle()` function to close the DCM handle that was created when the endpoint was published. If a server terminates, all published groups of server endpoints will also be unpublished.

To interrupt a blocking call of the `DcmListenGroupPub()` function from another thread, call the `DcmInterruptGroupPubBlockingCall()` function.

You can use the `DcmSubscribeToGroupPubEvent()` and `DcmUnsubscribeFromGroupPubEvent()` functions together with the functions from the notice_api.h API to receive notifications about the occurrence of the following events: the server received a request from the client to create an IPC channel (the `DCM_CLIENT_CONNECTED` flag), or a blocking call of the `DcmListenGroupPub()` function was interrupted (the `DCM_BLOCKING_CALL_INTERRUPTED` flag). (Flags of the event mask and the "resource–event mask" entry ID `DCM_EVENT_ID` are defined in the header file `sysroot-*-kos/include/dcm/dcm_api.h` from the KasperskyOS SDK.) The `DcmSubscribeToGroupPubEvent()` and `DcmUnsubscribeFromGroupPubEvent()` functions let you configure the notification receiver by adding or deleting, respectively, tracked objects identified by DCM handles.

pub.h functions

Function	Information about the function
`DcmPublishGroupEndpoints()`	Purpose Publishes a group of endpoints to `DCM`. Parameters • [in] `endpoints` – pointer to the array of structures containing information about grouped endpoints, such as the pointers to the qualified names of endpoints, pointers to the names of endpoint interfaces, and the contexts of endpoints. • [in] `endpointsCount` – number of grouped endpoints. • [in] `serverHandle` – handle that identifies the server. • [out] `outPubHandle` – pointer to the DCM handle that enables receipt of client requests to create IPC channels to use endpoints in a specific group with specific interfaces. Returned values

	If successful, the function returns `rcOk`, otherwise it returns an error code.
`DcmListenGroupPub()`	Purpose Receives a client request to create an IPC channel Parameters • [in] `pubHandle` – DCM handle obtained by calling the `DcmPublishGroupEndpoints()` function. • [in] `msec` – timeout (in milliseconds) for the appearance of a client request, or `INFINITE_TIMEOUT` to define an unlimited timeout. • [out] `outContext` – pointer to the context of an endpoint in the group. • [out] `outConnectionId` – pointer to the DCM handle that enables you to accept or reject a client request to create an IPC channel. • [out,optional] `outClientHandle` – pointer to the handle identifying the client, or `RTL_NULL` if this handle is not required. • [out,optional] `outClientName` – pointer to the buffer for the client name, or `RTL_NULL` if this name is not required. • [in] `clientNameSize` – size (bytes) of the buffer for the client name, or `0` if this name is not required. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If a blocking call was interrupted by calling the `DcmInterruptGroupPubBlockingCall()` function, `rcAbort` is returned. Additional information Non-blocking call if the `msec` parameter is set to `0`.
`DcmInterruptGroupPubBlockingCall()`	Purpose Interrupts a blocking call of the `DcmListenGroupPub()` function. Parameters • [in] `pubHandle` – DCM handle specified when calling the `DcmListenGroupPub()` function. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`DcmCloseGroupPubHandle()`	Purpose Closes the DCM handle that was received by calling the `DcmPublishGroupEndpoints()` function. Parameters • [in] `pubHandle` – DCM handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`DcmSubscribeToGroupPubEvent()`	Purpose Configures the notification receiver to receive notifications about events related to the object identified by the defined DCM handle. Parameters • [in] `notice` – identifier of the notification receiver.

	• [in] `pubHandle` – DCM handle obtained by calling the `DcmPublishGroupEndpoints()` function. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`DcmUnsubscribeFromGroupPubEvent()`	Purpose Configures the notification receiver to not receive notifications about events related to the object identified by the defined DCM handle. Parameters • [in] `notice` – identifier of the notification receiver. • [in] `pubHandle` – DCM handle obtained by calling the `DcmPublishGroupEndpoints()` function. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

Using the additional API for clients (subscr.h)

The API allows clients to receive notifications about the publishing and unpublishing of endpoint groups. From the client perspective, a group of endpoints is a set of endpoints on one server for which clients can use a single DCM handle to receive notifications about the publishing and unpublishing of those endpoints. However, there is a restriction that prevents a server from providing multiple endpoints with one interface. Multiple servers may provide identical groups of endpoints required by a client. A server may provide multiple groups of endpoints required by a client. A set of endpoints from different groups required by a client may overlap.

Information about API functions is provided in the table below.

To create a DCM handle that enables receipt of notifications about the publishing and unpublishing of endpoints with specific interfaces, call the `DcmSetGroupSubscription()` function. Use the `endpointTypes` parameter to define the names of interfaces of grouped endpoints. In the `endpointsCount` parameter, specify a number of grouped endpoints equal to the number of interfaces defined via the `endpointTypes` parameter. (Notifications will contain a number of endpoints equal to the number of interfaces because of the restriction dictating that one separate endpoint corresponds to each interface.)

To extract a notification about the publishing or unpublishing of an endpoint group from the notification queue, call the `DcmReadGroupPubQueue()` function. In the `subscrHandle` parameter of the `DcmReadGroupPubQueue()` function, specify the DCM handle that was obtained by calling the `DcmSetGroupSubscription()` function. In the `maxCount` parameter of the `DcmReadGroupPubQueue()` function, specify the number of elements in the array defined via the `outEndpoints` parameter. The value of this parameter must be greater than or equal to the number of endpoints in the group. The value obtained via the `outPubStatus` parameter of the `DcmReadGroupPubQueue()` function indicates that the group of endpoints was published or unpublished. A notification about the publishing of an endpoint group appears in the queue only if the server publishes all endpoints in the group together or one by one. If the server publishes only a portion of the endpoints required by the client, a notification about the publishing of an endpoint group does not appear. If the server terminates or unpublishes at least one endpoint in a published group of endpoints, the client receives a notification about the unpublishing of this endpoint group.

If you no longer need to receive notifications about the publishing and unpublishing of an endpoint group with specific interfaces, the DCM handle that was created by calling the `DcmSetGroupSubscription()` function must be closed by calling the `DcmCloseGroupSubscrHandle()` function.

To interrupt a blocking call of the `DcmReadGroupPubQueue()` function from another thread, call the `DcmInterruptGroupSubscrBlockingCall()` function.

You can use the `DcmSubscribeToGroupSubscrEvent()` and `DcmUnsubscribeFromGroupSubscrEvent()` functions together with the functions from the notice_api.h API to receive notifications about the occurrence of the following events: the server published or depublished an endpoint in `DCM` (the `DCM_PUBLICATION_CHANGED` flag), or a blocking call of the `DcmReadGroupPubQueue()` function was interrupted (the `DCM_BLOCKING_CALL_INTERRUPTED` flag). (Flags of the event mask and the "resource–event mask" entry ID `DCM_EVENT_ID` are defined in the header file `sysroot-*-kos/include/dcm/dcm_api.h` from the KasperskyOS SDK.) The `DcmSubscribeToGroupSubscrEvent()` and `DcmUnsubscribeFromGroupSubscrEvent()` functions let you configure the notification receiver by adding or deleting, respectively, tracked objects identified by DCM handles.

subscr.h functions

Function	Information about the function
`DcmSetGroupSubscription()`	Purpose Creates a DCM handle that enables receipt of notifications about the publishing and unpublishing of endpoint groups with specific interfaces. Parameters • [in] `endpointTypes` – pointer to the array containing the names of interfaces of the endpoints being grouped. • [in] `endpointsCount` – number of grouped endpoints. It must match the number of interfaces defined via the `endpointTypes` parameter. • [in,optional] `serverName` – pointer to the server name, or `RTL_NULL` to not use notification filtering based on the server name. • [in,optional] `serverHandle` – handle identifying the server, or `INVALID_HANDLE` to not use notification filtering based on a specific server. • [out] `outSubscrHandle` – pointer to the DCM handle that enables receipt of notifications about the publishing and unpublishing of endpoint groups with specific interfaces. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`DcmReadGroupPubQueue()`	Purpose Extracts a notification about the publishing or unpublishing of an endpoint group from the notification queue. Parameters • [in] `subscrHandle` – DCM handle that was obtained by calling the `DcmSetGroupSubscription()` function. • [in] `msec` – time-out before notifications appearing in milliseconds, or `INFINITE_TIMEOUT` to set an unlimited time-out. • [in] `maxCount` – number of elements in the array defined via the `outEndpoints` parameter. It must be greater than or equal to the number of endpoints in the group. • [out,optional] `outServerHandle` – pointer to the handle identifying the server, or `RTL_NULL` if this handle is not required. • [out,optional] `outServerName` – pointer to the buffer for the server name, or `RTL_NULL` if this name is not required. • [in] `serverNameSize` – size (bytes) for the server name, or `0` if this name is not required. • [out] `outEndpoints` – pointer to the array of structures containing the qualified names of endpoints in the group and the names of interfaces of the group endpoints. • [out] `outEndpointsCount` – pointer to the number of endpoints in the group.

	• [out] `outPubStatus` – pointer to the value indicating that the group of endpoints was published (`DcmEndpointPublished`) or unpublished (`DcmEndpointUnpublished`). Returned values If successful, the function returns `rcOk`, otherwise it returns an error code. If a blocking call was interrupted by calling the `DcmInterruptGroupSubscrBlockingCall()` function, `rcAbort` is returned. Additional information Non-blocking call if the `msec` parameter is set to `0`.
`DcmInterruptGroupSubscrBlockingCall()`	Purpose Interrupts a blocking call of the `DcmReadGroupPubQueue()` function. Parameters • [in] `subscrHandle` – DCM handle specified when calling the `DcmReadGroupPubQueue()` function. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`DcmCloseGroupSubscrHandle()`	Purpose Closes the DCM handle that was obtained by calling the `DcmSetGroupSubscription()` function. Parameters • [in] `subscrHandle` – DCM handle. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`DcmSubscribeToGroupSubscrEvent()`	Purpose Configures the notification receiver to receive notifications about events related to the object identified by the defined DCM handle. Parameters • [in] `notice` – identifier of the notification receiver. • [in] `subscrHandle` – DCM handle that was obtained by calling the `DcmSetGroupSubscription()` function. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.
`DcmUnsubscribeFromGroupSubscrEvent()`	Purpose Configures the notification receiver to not receive notifications about events related to the object identified by the defined DCM handle. Parameters • [in] `notice` – identifier of the notification receiver. • [in] `subscrHandle` – DCM handle that was obtained by calling the `DcmSetGroupSubscription()` function. Returned values If successful, the function returns `rcOk`, otherwise it returns an error code.

MessageBus component

The `MessageBus` component implements the message bus that ensures receipt, distribution and delivery of messages between programs running KasperskyOS. This bus is based on the publisher-subscriber model. Use of a message bus lets you avoid having to create a large number of IPC channels to connect each subscriber program to each publisher program.

> Messages transmitted through the `MessageBus` cannot contain data. These messages can be used only to notify subscribers about events.

The `MessageBus` component provides an additional level of abstraction over KasperskyOS IPC that helps simplify the development and expansion of your programs. `MessageBus` is a separate program that is accessed through IPC. However, developers are provided with a `MessageBus` access library that lets you avoid direct use of IPC calls. To get access to the API of this library, link your program to the library by using the CMake command `target_link_libraries()`.

CMakeLists.txt

```
target_link_libraries (${TARGET_NAME} ${messagebus_CLIENT_LIB})
```

The API of the access library provides the following interfaces:

`IProviderFactory`
: Interface that provides factory methods for obtaining access to instances of all other interfaces.

`IProviderControl`
: Interface for registering and deregistering a publisher and subscriber in the bus.

`IProvider (MessageBus component)`
: Interface for transferring a message to the bus.

`ISubscriber`
: Callback interface for sending a message to a subscriber.

`IWaiter`
: Interface for waiting for a callback when the corresponding message appears.

Message structure

Each message contains two parameters:

`topic`
: Identifier of the message subject.

`id`
: Additional parameter that identifies a particular message.

The `topic` and `id` parameters are unique for each message. The interpretation of `topic+id` is determined by the contract between the publisher and subscriber. For example, if there are changes to the configuration data used by the publisher and subscriber, the publisher forwards a message regarding the modified data and the `id` of the specific entry containing the new data. The subscriber uses mechanisms outside of the `MessageBus` to receive the new data based on the `id` key.

For an example of using the `MessageBus` component in a KasperskyOS-based solution, please refer to the Messagebus example section.

IProviderFactory interface

The `IProviderFactory` interface provides factory methods for receiving the interfaces necessary for working with the `MessageBus` component.

A description of the `IProviderFactory` interface is provided in the file `/opt/KasperskyOS-Community-Edition-<version>/sysroot-*-kos/include/messagebus/i_messagebus_control.h`.

An instance of the `IProviderFactory` interface is obtained by using the free `InitConnection()` function, which receives the name of the IPC channel between your application and the `MessageBus` program. The connection name is defined in the `init.yaml.in` file when describing the solution configuration. If the connection is successful, the output parameter contains a pointer to the `IProviderFactory` interface.

- The interface for registering and deregistering (see "IProviderControl interface") publishers and subscribers in the message bus is obtained by using the `IProviderFactory::CreateBusControl()` method.
- The interface containing the methods enabling the publisher to send messages to the bus (see "IProvider interface (MessageBus component)") is obtained by using the `IProviderFactory::CreateBus()` method.
- The interfaces containing the methods enabling the subscriber to receive messages from the bus (see "ISubscriber, IWaiter and ISubscriberRunner interfaces") are obtained by using the `IProviderFactory::CreateCallbackWaiter` and `IProviderFactory::CreateSubscriberRunner()` methods.

  > It is not recommended to use the `IWaiter` interface, because calling a method of this interface is a locking call.

i_messagebus_control.h (fragment)

```
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);
```

IProviderControl interface

The `IProviderControl` interface provides the methods for registering and deregistering publishers and subscribers in the message bus.

A description of the `IProviderControl` interface is provided in the file `/opt/KasperskyOS-Community-Edition-<version>/sysroot-*-kos/include/messagebus/i_messagebus_control.h`.

The `IProviderFactory` interface is used to obtain an interface instance.

Registering and deregistering a publisher

The `IProviderControl::RegisterPublisher()` method is used to register the publisher in the message bus. This method receives the message subject and puts the unique ID of the bus client into the output parameter. If the message subject is already registered in the bus, the call will be declined and the client ID will not be filled.

The `IProviderControl::UnregisterPublisher()` method is used to deregister a publisher in the message bus. This method accepts the bus client ID received during registration. If the indicated ID is not registered as a publisher ID, the call will be declined.

i_messagebus_control.h (fragment)

```
class IProviderControl
{
...
    virtual fdn::ResultCode RegisterPublisher(const Topic& topic, ClientId& id) = 0;
    virtual fdn::ResultCode UnregisterPublisher(ClientId id) = 0;
...
};
```

Registering and deregistering a subscriber

The `IProviderControl::RegisterSubscriber()` method is used to register the subscriber in the message bus. This method accepts the subscriber name and the list of subjects of messages for the necessary subscription, and puts the unique ID of the bus client into the output parameter.

The `IProviderControl::UnregisterSubscriber()` method is used to deregister a subscriber in the message bus. This method accepts the bus client ID received during registration. If the indicated ID is not registered as a subscriber ID, the call will be declined.

i_messagebus_control.h (fragment)

```
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;
...
};
```

IProvider interface (MessageBus component)

The `IProvider` interface provides the methods enabling the publisher to send messages to the bus.

A description of the `IProvider` interface is provided in the file `/opt/KasperskyOS-Community-Edition-<version>/sysroot-*-kos/include/messagebus/i_messagebus.h`.

The `IProviderFactory` interface is used to obtain an interface instance.

Sending a message to the bus

The `IProvider::Push()` method is used to send a message. This method accepts the bus client ID received during registration and the message ID. If the message queue in the bus is full, the call will be declined.

i_messagebus.h (fragment)

```
class IProvider
{
public:
...
    virtual fdn::ResultCode Push(ClientId id, BundleId dataId) = 0;
...
};
```

ISubscriber, IWaiter and ISubscriberRunner interfaces

The `ISubscriber`, `IWaiter`, and `ISubscriberRunner` interfaces provide the methods enabling the subscriber to receive messages from the bus and process them.

Descriptions of the `ISubscriber`, `IWaiter` and `ISubscriberRunner` interfaces are provided in the file `/opt/KasperskyOS-Community-Edition-<version>/sysroot-*-kos/include/messagebus/i_subscriber.h`.

The `IProviderFactory` interface is used to obtain instances of the `IWaiter` and `ISubscriberRunner` interfaces. The implementation of the `ISubscriber` callback interface is provided by the subscriber application.

Receiving a message from the bus

You can use the `IWaiter::Wait()` or `ISubscriberRunner::Run()` method to switch a subscriber to standby mode, waiting for a message from the bus. These methods accept the bus client ID and the pointer to the `ISubscriber` callback interface. If the client ID is not registered, the call will be declined.

It is not recommended to use the `IWaiter` interface, because calling the `IWaiter::Wait()` method is a locking call.

The `ISubscriber::OnMessage()` method will be called when a message is received from the bus. This method accepts the message subject and message ID.

i_subscriber.h (fragment)

```
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;
};
```

LogRR component

The `LogRR` component provides a system for logging information about program operation. The component includes a server program to which other programs forward messages about their state, and auxiliary output channels used for logging. Messages are sent to the server by using the `logrr_clog` library (the `logrr_cpp` library is available for C++), which filters messages according to the log level and sends them to the server. The server relays messages to output channels. An output channel saves log messages to files.

In the SDK, the LogRR program APIs are represented by static and dynamic libraries linked to the programs whose operating information is logged:

- The `logrr_clog` library and the header file `sysroot-*-kos/include/component/logrr/clog/clog.h` are used in C-language programs.
- The `logrr_cpp` library and the header file `sysroot-*-kos/include/component/logrr/cpp/logger.h` are used in C++ programs.

Adding a log server and output channels to a solution

The `LogrrServer` program sends messages to one or more output channels. You can write log messages to files by using the output channel named `FsOutputChannel`.

Adding and configuring logging server

The command for creating and configuring the log server `create_logrr_server()` may contain the following parameters:

`LOG_LEVEL`

Sets the specified log level. The available values for `LOG_LEVEL` are `CRITICAL`, `ERROR`, `WARNING`, `INFO`, `DEBUG` and `TRACE`. By default, the log server handles messages with the Info level.

`CONFIG_PATH`

Specifies the path to the configuration file of the log server.

`RECV_CORE_LOGS`

Enables collection of kernel logs.

`WRITE_TO_CORE`

Enables forwarding of logs to the kernel. Does not exclude forwarding of logs to the log server or further to the output channel. Required for printing logs to UART.

`OUTPUT_CHANNELS`

Connects one or more specified output channels to the server.

`DENY_DYNAMIC_CONN_TYPES`

Enables rejection of dynamic connections to the log server.

`DENY_STATIC_CONN_TYPES`

Enables rejection of static connections to the log server.

To add the `LogrrServer` *program to a solution:*

Edit the root file containing the CMake commands for building the solution or the file containing the CMake commands for building the `Einit` initializing program:

CMakeLists.txt

```
find_package (logrr_server REQUIRED COMPONENTS ENTITY)
# ...
set (TASKS
    # ...
    ${logrr_server_ENTITY})
# ...
# Add the server that uses the output channel FsOutputChannel
create_logrr_server(
    LOG_LEVEL DEBUG
    WRITE_TO_CORE ON
    OUTPUT_CHANNELS
        logrr.FsOutputChannel)
# Adding a server and output channel to a solution
list(APPEND ENTITIES
    logrr_server::default_entity
    logrr_fs_output_channel::default_entity)
# Create static IPC channels for source programs to connect to the server
connect_to_logrr_server (ENTITIES ClientOne)

build_kos_qemu_image (kos-qemu-image
    CONNECTIONS_CFG ${INIT_YAML_FILE}
    SECURITY_PSL ${SECURITY_PSL_FILE}
    PACK_DEPS ON
    ${TASKS})
```

The `PACK_DEPS` option must be included when calling the `build_kos_qemu_image()` or `build_kos_hw_image()` command to include an automatic build of the libraries in the disk image.

Adding and configuring output channels

The command for creating and configuring the channel for printing logs to files `create_logrr_fs_output_channel()` may contain the following parameters:

`CONFIG_PATH`

Specifies the path to the configuration file of the output channel.

`SERVER_TARGET`

Specifies the log server to which the created output channel must connect.

`APP_LOG_DIR`

Specifies the directory for storing log files for programs. Required parameter.

`SYS_LOG_DIR`

Specifies the directory for storing system log files. If this parameter is not defined, system log files are not saved.

To add the `FsOutputChannel` *to a solution:*

Edit the root file containing the CMake commands for building the solution or the file containing the CMake commands for building the `Einit` initializing program:

CMakeLists.txt

```
find_package (logrr_fs_output_channel   REQUIRED COMPONENTS ENTITY)
# Add the FsOutputChannel that writes to the file
create_logrr_fs_output_channel (
    APP_LOG_DIR "/log/apps"
    SYS_LOG_DIR "/log/sys")
```

Creating IPC channels and allowing interaction between programs

The build system automatically creates IPC channels required for the log system to work, and generates a security policy description that allows interaction between processes.

To create a static connection to the log server, you can use the CMake command `connect_to_logrr_server (ENTITIES <names_of_programs_to_connect>)`.

Getting log entries when running in QEMU

The client libraries `logrr_clog` and `logrr_cpp` send messages to the kernel, which forwards them to the UART port. You can forward data from an emulated UART port to a standard I/O stream by specifying the `-serial stdio` flag when starting QEMU. In this case, log messages will be displayed in the terminal in which QEMU is running.

Specify the QEMU startup flags in the value of the QEMU_FLAGS parameter that is passed to the build_kos_qemu_image() command:

CMakeLists.txt

```
build_kos_qemu_image (kos-qemu-image
            # ...
            QEMU_FLAGS "-m 4096 -serial stdio"
            # ...
            )
```

Using macros to send messages to a log

The `LogRR` program provides macros for quickly accessing logging functions with a specific log level.

Example use of macros in a C-language program:

C

```
#include <component/logrr/clog/clog.h>
// ...
CLOG(CRITICAL, "Critical message. Exit code = %d", 404);
CLOG(ERROR, "Error message.  Error code = %d", 13);
CLOG(WARNING, "Warning message. Details: %s", "operation not permitted.");
CLOG(INFO, "Info message. Status: %s", "connected.");
CLOG(DEBUG, "Debug message.");
CLOG(TRACE, "Trace message.");
```

To use these macros, you need to link the program to the logrr_clog library.

Example use of macros in a C++ program:

C++

```
#include <component/logrr/cpp/logger.h>
// ...
LOG(CRITICAL, "Critical message. Exit code = {}", 404);
LOG(ERROR, "Error message. Error code = {}", 13);
LOG(WARNING, "Warning message. Details: {}", "operation not permitted.");
LOG(INFO, "Info message. Status: {}", "connected.");
LOG(DEBUG, "Debug message.");
LOG(TRACE, "Trace message.");
```

To use these macros, you need to link the program to the logrr_cpp library.

Advanced capabilities when sending messages to a log

In most cases, you can send messages to the log by simply using macros for quick access to logging functions.

This section describes the additional capabilities of the `logrr_cpp` library API: merging messages, condition-based message forwarding, and deferred forwarding of messages.

Merging messages

To incrementally build the text of a log entry from parts and then send the entry to the log with one call:

1. Link the program to the logrr_cpp library.
2. Include the header file `component/logrr/cpp/tools.h`.

   C++
   ```
   #include <component/logrr/cpp/tools.h>
   ```

3. Create an instance of the `LogIface` class.

   C++
   ```
   auto logger = LogIface(logrr::LogLevel::Warning);
   ```

4. Fill the message buffer with one or more calls of the `LogIface::Push()` method.

   C++
   ```
   logger.Push("a={}", a);
   // ...
   logger.Push(", b={}", b);
   // ...
   logger.Push(", c={}", c);
   ```

5. Send the previously received messages to the log by using the `LogIface::Flush()` method.

   C++
   ```
   logger.Flush();
   ```

Sending a message to the log based on a condition

To send a message to the log only when a specified condition is fulfilled:

1. Link the program to the logrr_cpp library.
2. Include the header file `component/logrr/cpp/tools.h`.

```C++
#include <component/logrr/cpp/tools.h>
```

3. Pass the logical value, one of the `LogLevel` enumerators, and the message text to the `LOG_IF()` macro.

```C++
LOG_IF(IsWorldBroken(), logrr::LogLevel::Critical, "World is broken!");
```

Deferred forwarding of messages to the log

The `LOG_DEFER()` macro declared in the header file `component/logrr/cpp/tools.h` lets you avoid duplicating code for logging in the `if`...`else` blocks.

To send a pre-formatted message to the log when exiting a function:

1. Link the program to the logrr_cpp library.
2. Include the header file `component/logrr/cpp/tools.h`.

```C++
#include <component/logrr/cpp/tools.h>
```

3. Add a `LOG_DEFER()` macro call to the beginning of the code for the function that changes the values that need to be sent to the log. Pass the `LogLevel` value, message formatting string and arguments to be inserted in the formatting string to the macro.

```C++
int DeferredLoggingExample()
{
    auto logger = LOG_DEFER(logrr::LogLevel::Info, "a={}, b={}", a, b);
    if (someCondition)
    {
        a = 1;
        b = 2;
        return -1;
    }
    else
    {
        a = 3;
        b = 4;
        return 0;
    }
}
```

In this example, a message is sent to the log when the `logger` object's destructor is called on exit from the `DeferredLoggingExample()` function. The values `a` and `b` at the time of exit from the function are inserted into the message formatting string.

Log levels

A log level is an attribute of each message in the logging system. The log level is configured through the server and is used when filtering messages in a client.

Critical

Critical malfunction in program operation that makes it impossible for the program to continue working.

Error

Program operating error that does not completely prevent the program from continuing to work. This level is applied for logging an unexpected termination of the current operation. It is not recommended to apply this level for logging erroneous user actions.

Warning

Problem situation that is anticipated and normally handled by the program.

Info (default)

Information about the current functional performance of the program without additional details. For example, this info can include switching to a new window in the user interface or successfully writing an entry in the database.

Debug

More detailed log level than Info. It provides more details on program events and may be useful for debugging.

Trace

The most detailed log level that is used for the most detailed tracking of program events. When this level is enabled, it can have a substantial impact on performance.

Each successive level includes all previous levels. For example, if a server is configured for the Warning level, it forwards messages with the Critical, Error and Warning levels to the log while ignoring messages with the Info, Debug and Trace levels.

The log level when the program sends a message can be defined by doing the following:

- Using the macro with the parameter corresponding to the selected level.
- Passing the value of the `LogrrLogLevel` or `LogLevel` enumerator when logging functions are called.

LogRR program API

The libraries of the `LogRR` program provide APIs for sending messages to the log through the log server and APIs for managing the log server.

logrr_clog library

The `logrr_clog` library provides the API for sending messages to the log.

> The `logrr_clog` library is intended for use in C-language programs. Use the `logrr_cpp` library in C++ programs.

To get access to the API of this library:

Link your program to the library by using the `target_link_libraries()` CMake command.

```
CMakeLists.txt

target_link_libraries (${TARGET_NAME} ${logrr_clog_LIB})
```

ClogLog() function and CLOG macro

The `ClogLog()` function writes to the log.

> The `ClogLog()` function is intended for use in C-language programs. In C++ programs, use the Logger::Log() function or LOG_* macros.

A description of the `ClogLog()` function and macros for quick access to it are provided in the file `/opt/KasperskyOS-Community-Edition-<version>/sysroot-*-kos/include/component/logrr/clog/clog.h`.

```
component/logrr/clog/clog.h (fragment)

__attribute__((format(printf, 6, 7)))
void ClogLog(const char *module, LogLevel level, const char *file, int line, const char *func, const char
*message, ...);

#define CLOG_TRACE_IMPL_EX(level, file, line, func, ...) \
    ClogLog(LOG_MODULE, level, file, line, func, __VA_ARGS__)

#define CLOG_TRACE_IMPL(level, ...) \
    CLOG_TRACE_IMPL_EX((level), \
                       __FILENAME__, \
                       __LINE__, \
                       __func__, \
                       __VA_ARGS__)

#define __CLOG_LEVEL_TRACE__    LogrrLogLevel_Trace
#define __CLOG_LEVEL_DEBUG__    LogrrLogLevel_Debug
#define __CLOG_LEVEL_INFO__     LogrrLogLevel_Info
#define __CLOG_LEVEL_WARNING__  LogrrLogLevel_Warning
#define __CLOG_LEVEL_ERROR__    LogrrLogLevel_Error
#define __CLOG_LEVEL_CRITICAL__ LogrrLogLevel_Critical

#define CLOG(level, ...) CLOG_TRACE_IMPL(__CLOG_LEVEL_ ## level ## __, __VA_ARGS__)
```

Parameters of the `ClogLog()` function:

`level`

Log level `LogrrLogLevel`.

`file`

File name.

`line`

Number of the line in the file.

`func`

Function name.

`message`

Message text or message text formatting string.

`...`

Parameters to be inserted into the `message` formatting string.

Macro for quick access to the ClogLog() function

Instead of calling the `ClogLog()` function, you can use the macro whose description is provided in the file `component/logrr/clog/clog.h`. This macros is variadic (takes a variable number of parameters), which lets you avoid specifying all parameters of the `ClogLog()` function. When calling this macro, you only need to specify the log level and message text or the message formatting string with the values of parameters. The applied log `level` is determined by the first parameter of the macro. An example of using this macro is provided in the section titled Using macros to send messages to a log.

LogrrLogLevel enumerated type

The enumerated type `LogrrLogLevel` contains fields whose names correspond to the log levels available in the `LogRR` program.

> The enumerated type `LogrrLogLevel` is intended for use in C-language programs. In C++ programs, use the `LogLevel` class.

`LogrrLogLevel` enumerators can be passed to the `ClogLog()` function to send a message with the specified log level.

A description of `LogrrLogLevel` enumeration is provided in the file `/opt/KasperskyOS-Community-Edition-<version>/sysroot-*-kos/include/component/logrr/core/log_level_c.h`.

component/logrr/core/log_level_c.h (fragment)

```
typedef enum LogrrLogLevel
{
    LogrrLogLevel_Critical, /*!< Critical errors when process can't continue to work.  */
    LogrrLogLevel_Error,    /*!< Some issues that allow to continue application execution. */
    LogrrLogLevel_Warning,  /*!< Some regular errors that application can handle. */
    LogrrLogLevel_Info,     /*!< Some useful information about application work/state. */
    LogrrLogLevel_Debug,    /*!< Detailed log to understand/fix some specific issues. */
    LogrrLogLevel_Trace,    /*!< To "trace" some function execution. May affect performance. */
} LogLevel;
```

The logrr_cpp library

The `logrr_cpp` library provides the API for sending messages to the log.

> The `logrr_cpp` library is intended for use in C++ programs. In C-language programs, use the `logrr_clog` library.

To get access to the API of this library:

Link your program to the library by using the `target_link_libraries()` CMake command.

```
CMakeLists.txt

target_link_libraries (${TARGET_NAME} ${logrr_cpp_LIB})
```

Log() method and LOG macro

The `logrr::Log()` method writes to the log.

Method parameters:

`logLevel`

Log level `LogLevel`.

`sourceLocation`

Object that determines the location in the source code (file name, function name, line number and column number). The `std::source_location` class is used in C++20, while the `std::experimental::source_location` class is used in earlier versions.

`format`

Message text formatting string.

`args`

Parameters to be inserted into the `format` formatting string.

> The `logrr::Log()` method is intended for use in C++ programs. In C-language programs, use the static function ClogLog() or CLOG macro.

A description of the `logrr::Log()` method and macros for quick access to it are provided in the file `/opt/KasperskyOS-Community-Edition-<version>/sysroot-*-kos/include/component/logrr/cpp/logger.h`.

```
component/logrr/cpp/logger.h (fragment)

namespace logrr {

namespace details {

__rtl_public
void PrintErrorMessage(std::string_view fileName, std::uint32_t line, std::string_view exMsg) noexcept;

} // namespace details

__rtl_public
void Init(std::string_view providerName) noexcept;

__rtl_public
```

```
bool ShouldLog(logrr::LogLevel logLevel) noexcept;

__rtl_public
void SendMessage(
    std::string_view module,
    LogLevel logLevel,
    std::string_view file,
    const uint32_t line,
    std::string_view function,
    std::string_view msg) noexcept;

template <typename... Args>
__rtl_public
void Log(
    logrr::LogLevel logLevel,
    kos::rtl::SourceLocation sourceLocation,
    std::string_view format,
    Args&&... args) noexcept
{
    Log(logLevel, sourceLocation.file_name(), sourceLocation.line(),
        sourceLocation.function_name(), format, std::forward<Args>(args)...);
}

template <typename... Args>
__rtl_public
void Log(
    std::string_view module,
    logrr::LogLevel logLevel,
    kos::rtl::SourceLocation sourceLocation,
    std::string_view format,
    Args&&... args) noexcept
{
    Log(module, logLevel, sourceLocation.file_name(), sourceLocation.line(),
        sourceLocation.function_name(), format, std::forward<Args>(args)...);
}

template <typename... Args>
__rtl_public
void Log(
    logrr::LogLevel logLevel,
    std::string_view file,
    const uint32_t line,
    std::string_view function,
    std::string_view format,
    Args&&... args) noexcept
{
    Log("", logLevel, file, line, function, format, std::forward<Args>(args)...);
}

template <typename... Args>
__rtl_public
void Log(
    std::string_view module,
    logrr::LogLevel logLevel,
    std::string_view file,
    const uint32_t line,
    std::string_view function,
    std::string_view format,
    Args&&... args) noexcept
{
    if (ShouldLog(logLevel))
    try
    {
        constexpr auto MaxLength = static_cast<std::size_t>(LOGRR_LOG_MESSAGE_MAX_SIZE - 1);
        if constexpr (!sizeof...(args))
        {
            SendMessage(module, logLevel, file, line, function, format.substr(0, MaxLength));
        }
        else
        {
            std::array<std::uint8_t, MaxLength> fmtBuf;

            const auto endIt{fmt::format_to_n(
                fmtBuf.begin(),
                MaxLength,
                format,
                std::forward<Args>(args)...).out};

            const std::string_view formattedStr{
                reinterpret_cast<const char*>(fmtBuf.data()),
                static_cast<std::size_t>(std::distance(fmtBuf.begin(), endIt))};
```

```
            SendMessage(module, logLevel, file, line, function, formattedStr);
        }
    }
    catch (const std::exception& e)
    {
        details::PrintErrorMessage(file, line, e.what());
    }
}

} // namespace logrr

namespace {

const std::string_view LOG_MODULE;

} // namespace
#define __CPPLOG_LEVEL_TRACE__    logrr::LogLevel::Trace
#define __CPPLOG_LEVEL_DEBUG__    logrr::LogLevel::Debug
#define __CPPLOG_LEVEL_INFO__     logrr::LogLevel::Info
#define __CPPLOG_LEVEL_WARNING__  logrr::LogLevel::Warning
#define __CPPLOG_LEVEL_ERROR__    logrr::LogLevel::Error
#define __CPPLOG_LEVEL_CRITICAL__ logrr::LogLevel::Critical

#define LOG_INIT(providerName) \
    logrr::Init(providerName)

#define CALL_LOG(level, ...) \
    logrr::Log(LOG_MODULE, level, kos::rtl::SourceLocation::current(), __VA_ARGS__)

#define CALL_LOG_SL(level, file, line, func, ...) \
    logrr::Log(LOG_MODULE, level, file, line, func, __VA_ARGS__)

#define LOG(level, ...) CALL_LOG(__CPPLOG_LEVEL_ ## level ## __, __VA_ARGS__)
#define LOG_SL(level, file, line, func, ...) CALL_LOG_SL(__CPPLOG_LEVEL_ ## level ## __, file, line, func,
__VA_ARGS__)
```

A macro for quick access to the Log() method

Instead of calling the static method `Logger::Log()`, you can use the macro whose description is provided in the file `component/logrr/cpp/logger.h`. This macro is variadic (takes a variable number of parameters), which lets you avoid specifying all parameters of the `Logger::Log()` method. When calling this macro, you only need to specify the log level and message formatting string with the values of parameters. The applied log level `logLevel` is determined by the first parameter of the macro: An example of using these macros is provided in the section titled Using macros to send messages to a log.

LogLevel class

The `LogLevel` class is an enumeration (`enum class`) and contains fields whose names correspond to the log levels available in the `LogRR` program.

> The `LogLevel` class is intended for use in C++ programs. In C-language programs, use the enumerated type `LogrrLogLevel`.

Values of `LogLevel` enumeration can be passed to the following:

- Static method `Log()` of the `Logger` class
- Static methods `ChangeLogLevel()` and `ChangeGlobalLogLevel()` of the `Controller` structure
- `LogIface` class constructor

A description of the `LogLevel` class is provided in the file `/opt/KasperskyOS-Community-Edition-<version>/sysroot-*-kos/include/component/logrr/core/log_level.h`.

component/logrr/core/log_level.h (fragment)

```
enum class LogLevel : int8_t
{
    Critical = 0,
    Error,
    Warning,
    Info,
    Debug,
    Trace
};
```

LogIface class

The `LogIface` class contains methods that let you incrementally build the text of a log entry from parts and then send the entry to the log with one call:

- The `SetLogLevel()` method is used to set the log level of sent messages.
- The `Push()` method is used to add text to a message.
- The `Flush()` method is used to send a message composed of one or more `Push()` method calls to the log.

  When the `~LogIface()` destructor is called, the message composed of `Push()` calls is also sent to the log if the message was not previously sent using `Flush()`.
- The `Log()` method is used to immediately send a separate message (without using the text composed of `Push()` calls).

To create a `LogIface` class instance, pass one of the `LogLevel` enumerators to the class constructor. An example of using functions of the `LogIface` class is provided in the "Merging messages" subsection under Advanced capabilities when sending messages to a log.

A description of the `LogIface` class is provided in the file `/opt/KasperskyOS-Community-Edition-<version>/sysroot-*-kos/include/component/logrr/cpp/tools.h`.

component/logrr/cpp/tools.h (fragment)

```
/**
 * @brief C++ provider interface for logrr for simple printing and queuing logs
 * */
class LogIface
{
    // ...

public:
    LogIface(logrr::LogLevel level, sl::source_location loc = sl::source_location::current()) : m_level(level),
m_defaultSourceLocation(loc)
    {}

    void SetLogLevel(logrr::LogLevel new_level)
    {
        m_level = new_level;
    }

    /**
     * @brief Append new format with args to current message template
     * */
    template <typename... Args>
    void Push(std::string_view frmt, Args&&... args)
    {
        AppendFormat(m_message, frmt, std::forward<Args>(args)...);
```

```
    }

    /**
     * @brief Log instantly through separate Log command
     * */
    template <typename... Args>
    void Log(FormatWithLocation frmt, Args&&... args)
    {
        Flush_(frmt.loc, frmt.frmt, std::forward<Args>(args)...);
    }

    /**
     * @brief Flush current message, built with Push, to Log
     * */
    void Flush(const sl::source_location& loc = sl::source_location::current())
    {
        Flush_(loc, m_message);
        m_message.clear();
    }

    ~LogIface()
    {
        if (!m_message.empty())
        {
            Flush_(m_defaultSourceLocation, m_message);
        }
    }

    // ...
};
```

ExecutionManager component

The API is defined in the header files located in the directory `sysroot-*-kos/include/component/execution_manager/` from the KasperskyOS SDK.

The ExecutionManager component usage scenario is described in the article titled [Starting a processes using the ExecutionManager system program](#).

execution_manager_proxy.h interface

The API is defined in the header file `sysroot-*-kos/include/component/execution_manager/kos_ipc/execution_manager_proxy.h` from the KasperskyOS SDK.

The interface contains the `CreateExecutionManager()` function for getting the pointer to the instance of the `IExecutionManager` interface that is required for working with the ExecutionManager component. Using the `cfg` input parameter, this function accepts configuration parameters in the `ExecutionManagerConfig` structure. All fields of this structure are optional.

```
struct ExecutionManagerConfig
{
    // Name of the IPC channel for connecting to the ExecutionManager process.
    char const* const mainConnection            = KOS_EXECMGR_CONN_MAIN;
    // Name of the endpoint that implements the IApplicationController interface.
    char const* const appControlInterface       = KOS_EXECMGR_IFACE_AC;
    // Name of the endpoint that implements the IStateProvider interface.
    char const* const appStateInterface         = KOS_EXECMGR_IFACE_AS;
    // Name of the endpoint that implements the ISystemController interface.
    char const* const systemControlInterface    = KOS_EXECMGR_IFACE_SC;
    // Name of the endpoint that implements the IProcessControl interface.
    char const* const processControlInterface   = KOS_EXECMGR_IFACE_PC;
    // ExecutionManager process class name.
    char const* const execmgrServerName         = KOS_EXECMGR_SERVER_NAME;
};
```

execution_manager_proxy.h functions

Function	Information about the function
`CreateExecutionManager()`	Purpose Gets the pointer to the instance of the `IExecutionManager` interface that is required for working with the ExecutionManager component. Parameters • [in] `cfg` – `ExecutionManagerConfig` structure containing the configuration settings for the connection to the `ExecutionManager` process. • [out] `emp` – pointer to the instance of the `IExecutionManager` interface. Returned values If successful, the function returns `kos::rtl::Ok`, otherwise it returns an error code.

Usage example:

client.cpp

```
#include <component/execution_manager/kos_ipc/execution_manager_proxy.h>

int main(int argc, const char *argv[])
{
    // ...
    execution_manager::ipc::ExecutionManagerConfig emCfg
    {
        // Name of the IPC channel for connecting to the ExecutionManager process.
        // It must match the MAIN_CONN_NAME value in the
        // CMakeLists.txt file for building the ExecutionManager component.
        .mainConnection             = "ExecMgrEntity",
// ExecutionManager process class name.
        .execmgrServerName          = "kl.ExecMgrEntity"
    };
    execution_manager::IExecutionManagerPtr emPtr;

    kos::Result res = CreateExecutionManager(emCfg, emPtr);
    if (res != kos::Ok)
    {
        std::cerr << fmt::format("[{}] Failed to create execution manager. res=0x{:x}\n", selfName, res);
        return EXIT_FAILURE;
    }
    // ...
}
```

IExecutionManager interface

The API is defined in the header file `sysroot-*-kos/include/component/execution_manager/i_execution_manager.h` from the KasperskyOS SDK.

The `IExecutionManager` interface lets you access pointers to the following interfaces:

- `IProcessControl` – interface that lets you start a process from an executable file installed by the PackageManager component from the KPA package in a KasperskyOS-based solution, and lets you receive information about the process (including information about its completion) and stop this process.
- `IApplicationController` – interface that lets you start a process from any executable file whose location in the file system is known, and lets you stop this process.
- `IStateProvider` – interface for receiving information about a process that was started using the `IApplicationController` interface.

- `ISystemController` – interface for managing the system.

Usage example:

client.cpp

```
int main(int argc, const char *argv[])
{
    // ...
    alm::execution_manager::IProcessControlPtr pc;
    res = emPtr->GetProcessController(pc);
    if (res != kos::Ok)
    {
        std::cerr << fmt::format("[{}] Failed to create process controller. res=0x{:x}\n", selfName, res);
        return EXIT_FAILURE;
    }

    execmgr::IApplicationControllerPtr ac;
    if (ptr->GetApplicationController(ac) != kos::Ok)
    {
        std::cerr << "Cannot get application controller" << std::endl;
        return EXIT_FAILURE;
    }

    execmgr::ISystemControllerPtr sc;
    if (ptr->GetSystemController(sc) != kos::Ok)
    {
        std::cerr << "Cannot get system controller" << std::endl;
        return EXIT_FAILURE;
    }

    // ...
}
```

IProcessControl interface

The API is defined in the header file `sysroot-*-kos/include/alm/execution_manager/i_process_control.h` from the KasperskyOS SDK.

The API lets you do the following:

- Start a process from an executable file installed by the PackageManager component from the KPA package in a KasperskyOS-based solution.
- Get information about process, including information about its termination.
- Stop a process.

Information about API functions is provided in the table below.

Starting a process

To start the process, call the `StartProcess()` function. Using the `opts` input parameter, this function accepts process startup parameters in the `StartOptions` structure. All fields of this structure are optional for initialization.

```
struct StartOptions
{
    // Handle that identifies the parent process.
    IpcHandle       parent{NK_INVALID_HANDLE, 0, NK_INVALID_HANDLE};
    // List of custom handles
```

```
    // that will be transferred to the process when it is started.
    UserHandleItems userHandles;
    // Command-line arguments.
    Strings         extraArgs;
    // Environment variables.
    Strings         extraEnvs;};
```

During initialization of the `StartOptions` structure, the `parent` handle transport container is populated with the default values, which indicate the absence of a parent process. You can create a process hierarchy by passing the handle of a parent process in this parameter to a started child process.

The `userHandles` vector is a component of the `StartOptions` structure and contains a list of custom handles that will be transferred to the process when it is started. Items of this list are pairs that are defined in the `UserHandleItem` structure. These pairs consist of a handle and its name. After a process is started, the handles transferred to this process can be found by using the `KosTaskLookupResource()` function based on the names of these handles.

```
struct UserHandleItem
{
    // Handle name.
    std::string id;
    // Handle.
    IpcHandle   handle;
};
```

The input parameters `pkgId` and `runConfigId` of the `StartProcess()` function are defined in the KPA package manifest. Their values may be received by using functions from the IPackageManifest interface of the PackageManager component.

Stopping a process

To stop the process, call the `StopProcess()` function. Using the `stopPolicy` input parameter, this function receives the process stop policy from the `StopPolicy` enumeration.

```
enum class StopPolicy : uint32_t
{
    // Send a stop signal to the process.
    Normal = 0,
    // Immediately stop the process.
    Forced,
};
```

Receiving information about a process

To request information about a process, call the `GetProcessState()` function. Using the `state` output parameter, this function returns information about a process in the `ProcessContext` structure.

```
struct ProcessContext
{
    // State of the process.
    ProcessState         state{};
    // System status of process termination.
    ExecutableExitStatus exitStatus{};
    // Common code for process termination.
    ExecutableExitCode   exitCode{};
    // Specific code for process termination.
    ExecutableExitReason exitReason{};
};
```

The `state` enumeration is a component of the `ProcessContext` structure and describes the state of a process.

```
enum class ProcessState : uint32_t
{
    // Service constant.
    None = 0,
    // Process created.
    Created,
    // Process initialized.
    Prepared,
    // Process started.
    Running,
    // Process stopped unexpectedly.
    Failed,
    // Process stopped, but data on the process is available
    // to the GetProcessState() function.
    Finished,
    // Process terminated and does not exist in the system.
    NotExisting,
};
```

The `exitStatus` enumeration is a component of the `ProcessContext` structure and describes the reason for stopping a process. The `ExecutableExitStatus` enumeration is defined in the header file `sysroot-*-kos/include/alm/execution_manager/execution_manager_types.h` from the KasperskyOS SDK.

```
enum class [[nodiscard]] ExecutableExitStatus : std::underlying_type_t<TaskExitStatus>
{
    // Process has not yet been started by the ExecutionManager component.
    ExitUninitialized = TaskExitStatus::TaskExitUninitialized,
    // Process stopped unexpectedly as a result of an unhandled exception.
    ExitUnexpected    = TaskExitStatus::TaskExitUnexpected,
    // Process terminated on its own, including
    // after receiving a stop signal via a call
    // of the StopProcess() function with the Normal value in the process stop policy.
    ExitNormal        = TaskExitStatus::TaskExitNormal,
    // Process stopped by external request, including via a call of the
    // StopProcess() function with the Forced value in the process stop policy.
    ExitTerminated    = TaskExitStatus::TaskExitTerminated,
};
```

The numeric `exitCode` parameter is a component of the `ProcessContext` structure and contains the return code for a process that stopped on its own, including after receiving a stop signal. The values for this return code are defined by the developer of the KasperskyOS-based solution.

The numeric `exitReason` parameter is a component of the `ProcessContext` structure and contains the return code that specifies the reason for an unsuccessful attempt to start the process, or `rcOk` if the startup was successful. Parameter is defined in the header file `sysroot-*-kos/include/rtl_cpp/retcode.h` from the KasperskyOS SDK. This file contains return codes that are common for the APIs of all solution components and their constituent parts (see Return codes).

i_process_control.h functions

Function	Information about the function
`StartProcess()`	Purpose Starts a process. Parameters • [in] `pkgId` – ID of the KPA package. • [in] `runConfigId` – ID of the startup configuration. • [in] `opts` – `StartOptions` structure containing the process startup parameters. • [out] `accessToken` – link to the handle that identifies the started process. Returned values

	If successful, the function returns `kos::rtl::Ok`, otherwise it returns an error code.
`StopProcess()`	Purpose Stops a process. Parameters • [in] `accessToken` – handle that identifies the started process. • [in] `stopPolicy` – process stop policy. Returned values If successful, the function returns `kos::rtl::Ok`, otherwise it returns an error code.
`GetProcessState()`	Purpose Requests information about a process. Parameters • [in] `accessToken` – handle that identifies the started process. • [out] `state` – link to the `ProcessContext` structure containing information about the process. Returned values If successful, the function returns `kos::rtl::Ok`, otherwise it returns an error code.

Usage example:

client.cpp

```
int main(int argc, const char *argv[])
{
    // ...

    alm::execution_manager::IProcessControl::StartOptions startOptions{};
    alm::execution_manager::IpcHandle procToken{};

    // ID of the KPA package.
    const std::string   packageId{"application.Application"};
    // Startup configuration ID.
    const std::string   runConfigId{"app"};

    kos::rtl::Result res = pc->StartProcess(packageId, runConfigId, startOptions, procToken);
    if (res != kos::rtl::Ok)
    {
        std::cerr << fmt::format("[{}] Failed to start process. res=0x{:x}\n", selfName, res);
        return EXIT_FAILURE;
    }

    alm::execution_manager::ProcessContext procContext{};

    res = pc->GetProcessState(procToken, procContext);
    if (res != kos::rtl::Ok)
    {
        std::cerr << fmt::format("[{}] Failed to get process state. res=0x{:x}\n", selfName, res);
        return EXIT_FAILURE;
    }

    if (procContext.state == alm::execution_manager::ProcessState::Running)
    {
        std::cerr << fmt::format("[{}] Process state is Running.\n", selfName);
    }

    res = pc->StopProcess(procToken, alm::execution_manager::StopPolicy::Forced);
    if (res != kos::rtl::Ok)
    {
        std::cerr << fmt::format("[{}] Failed to stop process. res=0x{:x}\n", selfName, res);
        return EXIT_FAILURE;
    }

    // ...
}
```

IApplicationController interface

This API is defined in the header file `sysroot-*-kos/include/component/execution_manager/i_application_control.h` from the KasperskyOS SDK.

The API lets you start a process from an executable file and stop this process.

Information about API functions is provided in the table below.

To start the process, call the `StartEntity()` function. Using the `info` input parameter, this function accepts process startup parameters in the `StartEntityInfo` structure. All fields of this structure are optional for initialization. Using the `resInfo` output parameter, this function returns the link to the `StartEntityResultInfo` structure containing the process startup results.

To stop a process, call the `ShutdownEntity()` or `StopEntity()` function. The `entId` input parameter is used by these functions to receive the handle that identifies the started process.

```
struct IApplicationController
{
    struct StartEntityInfo
    {
        // Process name.Unless otherwise specified, the process class name will be used.
        // If the process class name is not specified, the executable file name will be used.
        std::string              entityName;
        // Process class. Unless otherwise specified, the process name will be used.
        // If the process name is not specified, the executable file name will be used.
        std::string              eiid
        // Command-line arguments.
        std::vector<std::string> args
        // Environment variables.
        std::vector<std::string> envs;
        // Policy for restarting a process when it crashes. Available values:
        // EntityRestartPolicy::DoNotRestart - do not restart.
        // EntityRestartPolicy::AlwaysRestart - always restart.
        EntityRestartPolicy      restartPolicy { EntityRestartPolicy::DoNotRestart };
    };

    struct StartEntityResultInfo
    {
        // Security class assigned to the process.
        std::string eiid;
        // Handle that identifies the started process.
        EntityId    entId;
        // Security ID of the started process.
        Uid         sid;
        // Name of the started process.
        std::string taskName;
    };
};
```

i_application_control.h functions

Function	Information about the function
`StartEntity()`	Purpose Starts a process. Parameters • [in] `runPath` – path to the executable file to be started. • [in] `info` – structure containing the `StartEntityInfo` process startup parameters. • [out] `resInfo` – structure containing the `StartEntityResultInfo` process startup results. • [out] `containerPath` – path to the isolated data storage of a process.

	Returned values If successful, the function returns `kos::Ok`, otherwise it returns an error code.
`ShutdownEntity()`	Purpose Sends a termination signal to a process. Parameters • [in] `entId` – handle that identifies the running process. Returned values If successful, the function returns `kos::Ok`, otherwise it returns an error code.
`StopEntity()`	Purpose Immediately stops the execution of a process. Parameters • [in] `entId` – handle that identifies the running process. Returned values If successful, the function returns `kos::Ok`, otherwise it returns an error code.

Usage example:

client.cpp

```
int main(int argc, const char *argv[])
{
    // ...
    const fs::path appPath{"/application"};

    execmgr::IApplicationController::StartEntityResultInfo result;
    execmgr::IApplicationController::StartEntityInfo info;

    info.entityName = std::string{"application.Application"};
    info.eiid = std::string{"application.Application"};
    info.args = std::vector<std::string>{"1", "ARG1", "ARG2" , "ARG3"};
    info.envs = std::vector<std::string>{"ENV1=10", "ENV2=envStr"};

    std::cout << "Starting application from elf\n";

    if (ac->StartEntity(appPath, info, result) != kos::Ok)
    {
        std::cerr << "Cannot start application from " << appPath << std::endl;
        return EXIT_FAILURE;
    }

    std::cout << "Application started with process sid " << result.sid << "\n";

    auto AppId = result.entId;

    if (ac->StopEntity(AppId) != kos::Ok)
    {
        std::cerr << "Cannot stop process " << appPath << std::endl;
        return EXIT_FAILURE;
    }

   // ...
}
```

IStateProvider interface

This API is defined in the header file `sysroot-*-kos/include/component/execution_manager/i_state_control.h` from the KasperskyOS SDK.

The API lets you receive information about a process that was started by using the `StartEntity()` function from the IApplicationController API (see [IApplicationController interface]), including information about the reason for its termination.

Information about API functions is provided in the table below.

To get information about the process, call the `GetApplicationState()` function. The `entityId` input parameter is used by this function to receive the `EntityId`-type value that identifies the running process. Using the `appState` output parameter, this function returns information about a process in the `AppContext` structure. `AppContext` structure is defined in the header file `sysroot-*-kos/include/component/execution_manager/types.h` from the KasperskyOS SDK.

```
struct AppContext
{
    // State of the process.
    AppState                                    state{};
    // Reason for stopping a process.
    alm::execution_manager::ExecutableExitStatus exitStatus{};
    // Return code for a process
    // that stopped on its own.
    alm::execution_manager::ExecutableExitCode   exitCode{};
    // Return code that specifies the reason for
    // an unsuccessful attempt to start a process.
    alm::execution_manager::ExecutableExitReason exitReason{};
};
```

The `state` enumeration is a component of the `AppContext` structure and describes the state of a process. `State` enumeration is defined in the header file `sysroot-*-kos/include/component/execution_manager/types.h` from the KasperskyOS SDK.

```
enum class AppState : uint32_t
{
    None = 0,
    // Process started.
    Started,
    // Process restarted.
    Restarted,
    // Process was stopped by using the StopEntity() function
    // from the IApplicationController API.
    Stopped,
    // Process was stopped by using the ShutdownEntity() function
    // from the IApplicationController API.
    Completed,
    // Process stopped unexpectedly
    // as a result of an unhandled exception.
    Failed,
};
```

The `exitStatus` enumeration is a component of the `AppContext` structure and describes the reason for stopping a process. The `ExecutableExitStatus` enumeration is defined in the header file `sysroot-*-kos/include/alm/execution_manager/execution_manager_types.h` from the KasperskyOS SDK.

```
enum class [[nodiscard]] ExecutableExitStatus : std::underlying_type_t<TaskExitStatus>
{
    // Process has not yet been started by the ExecutionManager component.
    ExitUninitialized = TaskExitStatus::TaskExitUninitialized,
    // Process stopped unexpectedly as a result of an unhandled exception.
    ExitUnexpected    = TaskExitStatus::TaskExitUnexpected,
    // Process terminated on its own,
    // including after receiving a stop signal
    // via a call of the ShutdownEntity() function from the IApplicationController API.
    ExitNormal        = TaskExitStatus::TaskExitNormal,
    // Process stopped by external request, including via a call of
    // the StopEntity() function from the IApplicationController API.
    ExitTerminated    = TaskExitStatus::TaskExitTerminated,
};
```

The numeric `exitCode` parameter is a component of the `AppContext` structure and contains the return code for a process that stopped on its own, including after receiving a stop signal. The values for this return code are defined by the developer of the KasperskyOS-based solution.

The numeric `exitReason` parameter is a component of the `AppContext` structure and contains the return code that specifies the reason for an unsuccessful attempt to start the process, or `rcOk` if the process started successfully. Parameter is defined in the header file `sysroot-*-kos/include/rtl_cpp/retcode.h` from the KasperskyOS SDK. This file contains return codes that are common for the APIs of all solution components and their constituent parts (see Return codes).

i_state_control functions

Function	Information about the function
`GetApplicationState()`	Purpose Requests information about a process. Parameters • [in] `entityId` – value of the `EntityId` type that identifies the running process. • [out] `appState` – link to the `AppContext` structure containing information about the process. Returned values If successful, the function returns `kos::Ok`, otherwise it returns an error code.

Usage example:

client.cpp

```
int main(int argc, const char *argv[])
{
    // ...

    execution_manager::AppContext appContext();

    if (rc = m_sc->GetApplicationState(entityId, appContext), rc != kos::Ok)
    {
        KOS_LOGERROR("GetApplicationState: failure with entity {}: {}", entityName, RC_GET_CODE(rc));
        return rc;
    }

    // ...
}
```

ISystemController interface

The API is defined in the header file `sysroot-*-kos/include/component/execution_manager/i_system_control.h` from the KasperskyOS SDK.

The API lets you terminate the system.

Information about API functions is provided in the table below.

i_system_control.h functions

Function	Information about the function
`StopAllEntities()`	Purpose Stops all running processes, then terminates the ExecutionManager process, then sends a device shutdown request to the kernel. Parameters

	N/A Returned values If successful, the function returns `kos::Ok`, otherwise it returns an error code.

Usage example:

client.cpp

```
int main(int argc, const char *argv[])
{
 // ...

 if (sc->StopAllEntities() != kos::Ok)
 {
 std::cerr << "Cannot stop all processes\n";
 return EXIT_FAILURE;
 }

 // ...
}
```

PackageManager component

The API is defined in the header files located in the directory `sysroot-*-kos/include/component/package_manager/` from the KasperskyOS SDK.

The PackageManager component usage scenario is described in the article titled PackageManager component usage scenario.

The package_manager_proxy.h interface

This API is defined in the header file `sysroot-*-kos/include/component/package_manager/kos_ipc/package_manager_proxy.h` from the KasperskyOS SDK.

The interface contains function `CreatePackageManager()` for getting the pointer to the instance of the `IPackageManager` interface that is required for working with the PackageManager component. Using the `cfg` input parameter, this function accepts configuration parameters in the `PackageManagerConfig` structure. All fields of this structure are optional.

```
struct PackageManagerConfig
{
    // Name of the IPC channel for connecting to the PackageManager process.
    std::string const mainConnection = KOS_PKGMGR_CONN_MAIN;
    // Name of the endpoint that implements the IPackageController interface.
    std::string const pkgControlInterface = KOS_PKGMGR_IFACE_PKGC;
    // PackageManager process class name.
    std::string const pkgMgrServerName = KOS_PKGMGR_SERVER_NAME;
};
```

package_manager_proxy.h functions

Function	Information about the function
`CreatePackageManager()`	Purpose Gets the pointer to the instance of the `IPackageManager` interface that is required for working with the PackageManager component.

	Parameters • [in] `cfg`, `PackageManagerConfig` structure containing the configuration settings for the connection to the `PackageManager` process. • [out] `iPkgMgr` – pointer to the instance of the `IPackageManager` interface. Returned values If successful, the function returns `kos::rtl::Ok`, otherwise it returns an error code.

Usage example:

client.cpp

```
#include <component/package_manager/kos_ipc/package_manager_proxy.h>

namespace pkgmgr = package_manager;

int main(int argc, const char *argv[])
{
    // ...

    pkgmgr::IPackageManagerPtr iPkgMgr;
    std::string mainConnection{"PkgMgrEntity"};
    std::string pkgControlInterface{"kl.package_manager.PackageManager.ipkgc"};
    pkgmgr::ipc::PackageManagerConfig cfg{mainConnection, pkgControlInterface};

    if (CreatePackageManager(cfg, iPkgMgr) != kos::rtl::Ok)
        return EXIT_FAILURE;

    // ...
}
```

IPackageManager interface

The API is defined in the header file `sysroot-*-kos/include/component/package_manager/i_package_manager.h` from the KasperskyOS SDK.

The `IPackageManager` interface lets you receive the pointer to the `IPackageController` interface. This interface is intended for installing KPA packages to a KasperskyOS-based solution, and gets information about these packages.

i_package_manager.h functions

Function	Information about the function
`GetPackageController()`	Purpose Gets the pointer to an instance of the `IPackageController` interface. Parameters • [out] `pkgController` – pointer to the instance of the `IPackageController` interface. Returned values If successful, the function returns `kos::rtl::Ok`, otherwise it returns an error code.

Usage example:

client.cpp

```
int main(int argc, const char *argv[])
{
    // ...

    pkgmgr::IPackageControllerPtr pkgc;
```

```
    if (iPkgMgr->GetPackageController(pkgc) != kos::rtl::Ok)
        return EXIT_FAILURE;
    // ...
}
```

IPackageController interface

The API is defined in the header file `sysroot-*-kos/include/component/package_manager/i_package_control.h` from the KasperskyOS SDK.

The API lets you do the following:

- Install a KPA package in a KasperskyOS-based solution and remove the KPA package.
- Receive information about an installed KPA package: unique ID, data from the KPA package manifest, and the KPA package status (installed or removed).

Information about API functions is provided in the table below.

Installing a KPA package

To install a KPA package, call the `InstallPackage()` function. Using the `pkgInfo` input parameter, this function receives data to verify the certificates of an installed KPA package in the `InstallPackageInfo` structure. All fields of this structure are optional.

```
struct InstallPackageInfo {
    // Full name of the KPA package external signature file.
    std::filesystem::path signaturePath = "";
    // Full name of the KPA package index file.
    std::filesystem::path indexPath = "";
    // Indicates whether to verify certificates of the installed KPA package.
    bool signatureVerify = false
};
```

If the `signatureVerify` parameter is set to `true` (verify certificates of the installed KPA package) but the file names are not specified, the following default values will be used during installation of the KPA package: `<package_name>.kcat` for the KPA package external signature file and `<package_name>.kidx` for the KPA package index file.

Deleting a KPA package

To remove a KPA package, call the `UninstallPackage()` function.

Getting information about installed KPA packages

To get the unique IDs of installed KPA packages, call the `ListInstalledPackages()` function. Using the `pkgUIDs` output parameter, the function returns a list of unique IDs of installed packages.

To get data on the KPA package manifest, call the `GetManifest()` function. Using the `manifest` output parameter, the function returns the pointer of the instance of the `IPackageManifest` interface that can be used to access the KPA package manifest key values. For more details, refer to IPackageManifest interface.

Notifications about the statuses of KPA packages in a KasperskyOS-based solution

The PackageManager component implements a mechanism for notifications about the statuses of KPA packages in a KasperskyOS-based solution, thereby enabling you to track changes to their statuses. To receive notifications, you must create a notification queue by using the `CreateEventQueue()` function.

To extract notifications from this queue, you must use the `GetEvents()` function. Using the `pkgEvents` output parameter, the function returns the status of KPA packages (since the moment when the queue was created) in the `PackageEvent` structure.

```
struct PackageEvent
{
    enum class Type
    {
        // KPA package was installed.
        INSTALL,
        // KPA package was removed.
        UNINSTALL,
        // Notification queue overflowed.
        OVERFLOWED,
    };
    // KPA package status.
    Type type;
    // Unique ID of the KPA package.
    std::string uid;
};
```

To delete the notification queue, call the `DestroyEventQueue()` function.

i_package_control.h functions

Function	Information about the function
`InstallPackage()`	Purpose Installs a KPA package. Parameters • [in] `pkgName` – name of the KPA package to be installed. • [out] `pkgUID` – unique ID of the installed KPA package. • [in] `pkgInfo` – `InstallPackageInfo` structure that contains data for verifying certificates of an installed KPA package. Returned values If successful, the function returns `kos::rtl::Ok`, otherwise it returns an error code.
`UninstallPackage()`	Purpose Removes a KPA package. Parameters • [in] `pkgUID` – unique ID of the KPA package to be removed. Returned values If successful, the function returns `kos::rtl::Ok`, otherwise it returns an error code.
`ListInstalledPackages()`	Purpose Gets the unique IDs of installed KPA packages. Parameters • [out] `pkgUIDs` – array of unique IDs of installed KPA packages.

	Returned values If successful, the function returns `kos::rtl::Ok`, otherwise it returns an error code.
`GetManifest()`	Purpose Gets the pointer to the instance of the `IPackageManifest` interface required for receiving data on the KPA package manifest. Parameters • [in] `pkgUID` – unique ID of the KPA package whose manifest data is required. • [out] `manifest` – pointer to the instance of the `IPackageManifest` interface. Returned values If successful, the function returns `kos::rtl::Ok`, otherwise it returns an error code.
`GetItemDigest()`	Purpose Gets the checksum of the KPA package component file from the PackageManager component database. Parameters • [in] `itemPath` – full name of the KPA package component file whose checksum is required. • [out] `digest` – checksum of the KPA package component file. Returned values If successful, the function returns `kos::rtl::Ok`, otherwise it returns an error code.
`GetArtifactBlob()`	Purpose Gets the contents of a KPA package component. Parameters • [in] `pkgUID` – unique ID of the KPA package containing the requested component. • [in] `artifactType` – type of KPA package component. For more details, refer to List of "components" objects. • [in] `artifactName` – name of the KPA package component. • [out] `blob` – array of bytes representing the contents of the KPA package component. Returned values If successful, the function returns `kos::rtl::Ok`, otherwise it returns an error code.
`CreateEventQueue()`	Purpose Creates a queue for notifications about the statuses of KPA packages. Parameters • [out] `queueHandle` – link to the handle of the notification queue. Returned values If successful, the function returns `kos::rtl::Ok`, otherwise it returns an error code.
`DestroyEventQueue()`	Purpose Deletes the queue for notifications about the statuses of KPA packages. Parameters • [in] `queueHandle` – handle of the notification queue. Returned values

	If successful, the function returns `kos::rtl::Ok`, otherwise it returns an error code.
`GetEvents()`	Purpose Gets the statuses of KPA packages from the notification queue. Parameters • [in] `queueHandle` – handle of the notification queue. • [out] `pkgEvents` – `PackageEvent` structure containing the statuses of KPA packages. Returned values If successful, the function returns `kos::rtl::Ok`, otherwise it returns an error code.

Usage example:

client.cpp

```
#include <component/package_manager/kos_ipc/package_manager_proxy.h>

namespace pkgmgr = package_manager;

int main(int argc, const char *argv[])
{
    // ...

    // Create a queue for notifications about the statuses of KPA packages.
    Handle eventQueue;
    if (pkgc->CreateEventQueue(eventQueue) != kos::rtl::Ok)
        return EXIT_FAILURE;

    // Install the KPA package named test1.
    std::string uid1;
    const std::string pkgName1{"test1"};
    const package_manager::InstallPackageInfo pkgInfo1{"", "", true};

    if (pkgc->InstallPackage(pkgName1, uid1, pkgInfo1) != kos::rtl::Ok)
        return EXIT_FAILURE;

    // Install the KPA package named test2.
    std::string uid2;
    const std::string pkgName2{"test2"};
    const package_manager::InstallPackageInfo pkgInfo2{"/test2.kcat", "/test2.kidx", true};
    if (pkgc->InstallPackage(pkgName2, uid2, pkgInfo2) != kos::rtl::Ok)
        return EXIT_FAILURE;

    // Get the pointer to the instance of the IPackageManifest interface
    // for the KPA package named test1.
    pkgmgr::IPackageManifestPtr manifest;
    if (pkgc->GetManifest(uid1, manifest) != kos::rtl::Ok)
        return EXIT_FAILURE;

    // Get the unique IDs of installed KPA packages.
    std::vector<std::string> pkgUIDs
    if (pkgc->ListInstalledPackages(pkgUIDs) != kos::rtl::Ok)
        return EXIT_FAILURE;

    // Remove the KPA package named test1.
    if (pkgc->UninstallPackage(uid1) != kos::rtl::Ok)
        return EXIT_FAILURE;

    // Remove the KPA package named test2.
    if (pkgc->UninstallPackage(uid2) != kos::rtl::Ok)
        return EXIT_FAILURE;

    // Get the statuses of KPA packages from the notification queue.
    std::vector<package_manager::PackageEvent> pkgEvents;
    if (pkgc->GetEvents(eventQueue, pkgEvents) != kos::rtl::Ok)
        return EXIT_FAILURE;

    // Delete the notification queue.
    if (pkgc->DestroyEventQueue(eventQueue) != kos::rtl::Ok)
        return EXIT_FAILURE;

    if (KnHandleClose(eventQueue) != rcOk)
        return EXIT_FAILURE;
```

```
    // ...
}
```

IPackageManifest interface

This API is defined in the header file `sysroot-*-kos/include/component/package_manager/i_package_manifest.h` from the KasperskyOS SDK.

The API lets you get the KPA package manifest key values. For more details about the available keys of the KPA package manifest, see "KPA package manifest"

Information about API functions is provided in the table below.

Getting program ID

To get the ID of a program that was installed from a KPA package, call the `GetPackageID()` function.

Getting information about KPA package components

To get information about KPA package components, call the `GetApplicationsInfo()` function. Using the `applications` output parameter, the function returns information about KPA package components as a vector of `ApplicationInfo` structures.

```
struct ApplicationInfo
{
    // Name of the KPA package component.
    std::string name;
    // Path of the directory relative to the path /<package name>/res
    // in which the KPA package component is installed.
    std::string path;
    // Unused parameter.
    std::string iconPath;
    // Unused parameter.
    std::string containerPath;
    // Unused parameter.
    std::string etcPath;
    // Checksum of the KPA package component file.
    std::string digest;
    // Object in JSON format.
    ManifestType extensions;
    // Type of KPA package component.
    std::string type;
};
```

An `extensions` structure element describes an object in JSON format. `ManifestType` is an alias of the `nlohmann:json` type.

Getting program startup configurations

To get information about the startup configurations of a program installed from a KPA package, call the `GetRunConfigurationsInfo()` function. Using the `runConfigurations` output parameter, the function returns information about program startup configurations as a vector of `RunConfigurationInfo` structures.

```
struct RunConfigurationInfo
{
    // Startup configuration ID that is unique within the specific KPA package.
    std::string id;
    // Startup configuration name.
    std::string name;
    // Program security class.
    std::string eiid
    // Path to the KPA package component file.
    std::string path;
    // List of arguments in the form of a string array.
    std::vector<std::string> args{}
    // List of environment variables.
    std::vector<std::string> envs{}
    // Indicates whether the startup configuration is the primary one during program startup:
    // true means that it is primary, and false means that it is not primary.
    bool primary{false};
    // Indicates whether this configuration is started automatically:
    // true means that it is started automatically, and false means that it is not.
    bool autorun{false};
};
```

Receiving information about a program

To get information about a program installed from a KPA package, call the `GetPackageInfo()` function. Using the `packageInfo` output parameter, the function returns information about startup configurations in the `PackageInfo` structure.

```
struct PackageInfo
{
    // Unique ID of the program.
    std::string id;
    // Program name.
    std::string name;
    // Program version.
    std::string version;
    // Unused parameter.
    std::string title;
    // Unused parameter.
    std::string summary;
    // Program description.
    std::string description;
    // Program build number.
    std::string buildNumber;
    // Logical value indicating whether the program is a system program:
    // true means that it is, while false means that it is not.
    bool systemApplication;
    // Object of an arbitrary format.
    ManifestType extensions;
};
```

Getting information about the isolated storage of a program

To get information about the isolated storage of a program, call the `GetPrivateStorageInfo()` function. Using the `packageInfo` output parameter, the function returns information about startup configurations in the `PrivateStorage` structure.

```
struct PrivateStorage
{
    // Program data storage size in MB.
    size_t size;
    // Program data storage file system type.
    std::string fsType;
};
```

Getting information about objects of an arbitrary format

To get information about objects of an arbitrary format that were added by the KPA package developer (the `extensions` key from the KPA package manifest), call the `Get()` function.

i_package_manifest.h functions

Function	Information about the function
`GetPackageID()`	Purpose Gets the unique ID of a program installed from a KPA package. Parameters • [out] `id` – unique ID of the program. Returned values If successful, the function returns `kos::rtl::Ok`, otherwise it returns an error code.
`GetApplicationsInfo()`	Purpose Gets information about KPA package components. Parameters • [out] `applications` – vector of `ApplicationInfo` structures containing information about KPA package components. Returned values If successful, the function returns `kos::rtl::Ok`, otherwise it returns an error code.
`GetRunConfigurationsInfo()`	Purpose Gets information about the startup configurations of a program installed from a KPA package. Parameters • [out] `runConfigurations` – vector of `RunConfigurationInfo` structures containing information about startup configurations. Returned values If successful, the function returns `kos::rtl::Ok`, otherwise it returns an error code.
`GetPackageInfo()`	Purpose Gets information about a program installed from a KPA package. Parameters • [out] `packageInfo` – `PackageInfo` structure containing information about the program. Returned values If successful, the function returns `kos::rtl::Ok`, otherwise it returns an error code.
`GetPrivateStorageInfo()`	Purpose Gets information about the isolated storage of a program. Parameters • [out] `packageInfo` – `PrivateStorage` structure containing information about the isolated storage of a program. Returned values If successful, the function returns `kos::rtl::Ok`, otherwise it returns an error code.
`Get()`	Purpose

	Gets information about objects of an arbitrary format that were added to the manifest by the KPA package developer. Parameters N/A Returned values Returns the pointer of an object with the type `pkgmgr::IPackageManifest::ManifestType`.

Usage example:

client.cpp

```
#include <component/package_manager/kos_ipc/package_manager_proxy.h>

namespace pkgmgr = package_manager;

int main(int argc, const char *argv[])
{
    // ...

    // Get the pointer to the instance of the IPackageManifest interface
    // for the KPA package containing the unique ID uid1.
    pkgmgr::IPackageManifestPtr manifest;
    if (pkgc->GetManifest(uid1, manifest) != kos::rtl::Ok)
        return EXIT_FAILURE;

    // Get unique ID of the program.
    std::string packageID;
    if (manifest->GetPackageID(packageID) != kos::rtl::Ok)
        return EXIT_FAILURE;

    std::cout << packageID << '\n';

    // Get information about KPA package components.
    std::vector<package_manager::IPackageManifest::ApplicationInfo> infos;
    manifest->GetApplicationsInfo(infos);

    for (auto &info: infos)
    {
        std::cout << info.name << " " << info.path << "\n";
    }

    // Get information about objects of an arbitrary format.
    pkgmgr::IPackageManifest::ManifestType jsonManifest = manifest->Get();

    std::cout << jsonManifest["package"]["description"] << '\n';

    // Get information about the startup configurations of a program installed from a KPA package.
    std::vector<package_manager::IPackageManifest::RunConfigurationInfo> runConfigurationInfos;
    manifest->GetRunConfigurationsInfo(runConfigurationInfos);

    for (auto &runConfigurationInfo : runConfigurationInfos)
    {
        std::cout << "RunConfiguration.id = " << runConfigurationInfo.id << ", autorun = " <<
runConfigurationInfo.autorun << "\n";
    }

    // ...
}
```

Building a KasperskyOS-based solution

This section contains the following information:

- Description of the KasperskyOS-based solution build process.
- Descriptions of the scripts, libraries and build templates provided in KasperskyOS Community Edition.
- Information on how to use dynamic libraries in a KasperskyOS-based solution.

Building a solution image

A *KasperskyOS-based solution* consists of system software (including the KasperskyOS kernel and Kaspersky Security Module) and application software integrated for operation within a software/hardware system.

For more details, refer to Structure and startup of a KasperskyOS-based solution.

System programs and application software

Programs are divided into two types according to their purpose:

- *System programs* create the infrastructure for application software. For example, they facilitate hardware operations, support the IPC mechanism, and implement file systems and network protocols. System programs are included in KasperskyOS Community Edition. If necessary, you can develop your own system programs.
- *Application software* is designed for interaction with a solution user and for performing user tasks. Application software is not included in KasperskyOS Community Edition.

Building programs during the solution build process

During a solution build, programs are divided into the following two types:

- System programs provided as executable files in KasperskyOS Community Edition.
- System programs or application software that requires linking to an executable file.

Programs that require linking are divided into the following types:

- System programs that implement an IPC interface whose ready-to-use transport libraries are provided in KasperskyOS Community Edition.
- Application software that implements its own IPC interface. To build this software, transport methods and types need to be generated by using the NK compiler.
- Client programs that do not provide endpoints.

Building a solution image

KasperskyOS Community Edition provides an image of the KasperskyOS kernel and the executable files of some system programs and driver applications that are ready to use in a solution.

A specialized Einit program intended for starting all other programs, and a Kaspersky Security Module are built for each specific solution and are therefore not already provided in KasperskyOS Community Edition. Instead, the toolchain provided in KasperskyOS Community Edition includes the tools for building these resources.

The general step-by-step build scenario is described in the article titled Build process overview. A solution image can be built as follows:

- **[Recommended]** Using scripts of the `CMake` build system, which is provided in KasperskyOS Community Edition.
- [Without CMake] Using other automated build systems or manually with scripts and compilers provided in KasperskyOS Community Edition.

Build process overview

To build a solution image, the following is required:

1. Prepare EDL, CDL and IDL descriptions of applications, an init description file (`init.yaml` by default), and files containing a description of the solution security policy (`security.psl` by default).

   When building with `CMake`, an EDL description can be generated by using the `generate_edl_file()` command.

2. Generate *.edl.h files for all programs except the system programs provided in KasperskyOS Community Edition.

   - When building with `CMake`, the `nk_build_edl_files()` command is used for this purpose.
   - When building without `CMake`, the NK compiler must be used for this.

3. For programs that implement their own IPC interface, generate code of the transport methods and types that are used for generating, sending, receiving and processing IPC messages.

   - When building with `CMake`, the `nk_build_idl_files()` and `nk_build_cdl_files()` commands are used for these purposes.
   - When building without `CMake`, the NK compiler must be used for this.

4. Build all programs that are part of the solution, and link them to the transport libraries of system programs or applications if necessary. To build applications that implement their own IPC interface, you will need the code containing transport methods and types that was generated at step 3.

   - When building with `CMake`, standard build commands are used for this purpose. The necessary cross-compilation configuration is done automatically.
   - When building without `CMake`, the cross compilers included in KasperskyOS Community Edition must be manually used for this purpose.

5. Build the Einit initializing program.

- When building with `CMake`, the `Einit` program is built during the solution image build process using the `build_kos_qemu_image()` and `build_kos_hw_image()` commands.

- When building without `CMake`, the einit tool must be used to generate the code of the `Einit` program. Then the `Einit` application must be built using the cross compiler that is provided in KasperskyOS Community Edition.

6. Build the Kaspersky Security Module.

   - When building with `CMake`, the security module is built during the solution image build process using the `build_kos_qemu_image()` and `build_kos_hw_image()` commands.

   - When building without `CMake`, the `makekss` script must be used for this purpose.

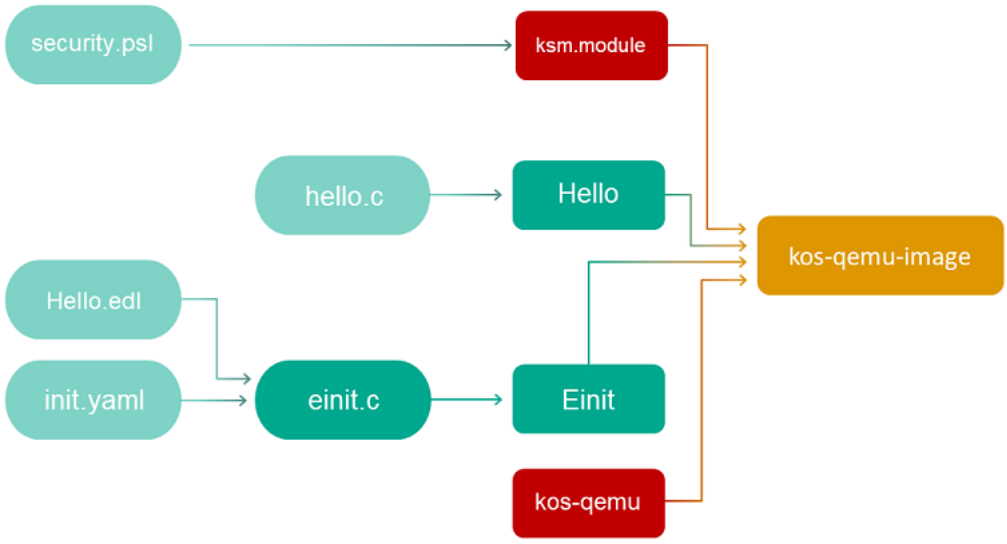
7. Create the solution image.

   - When building with `CMake`, the `build_kos_qemu_image()` and `build_kos_hw_image()` commands are used for this purpose.

   - When building without `CMake`, the `makeimg` script must be used for this.

Example 1

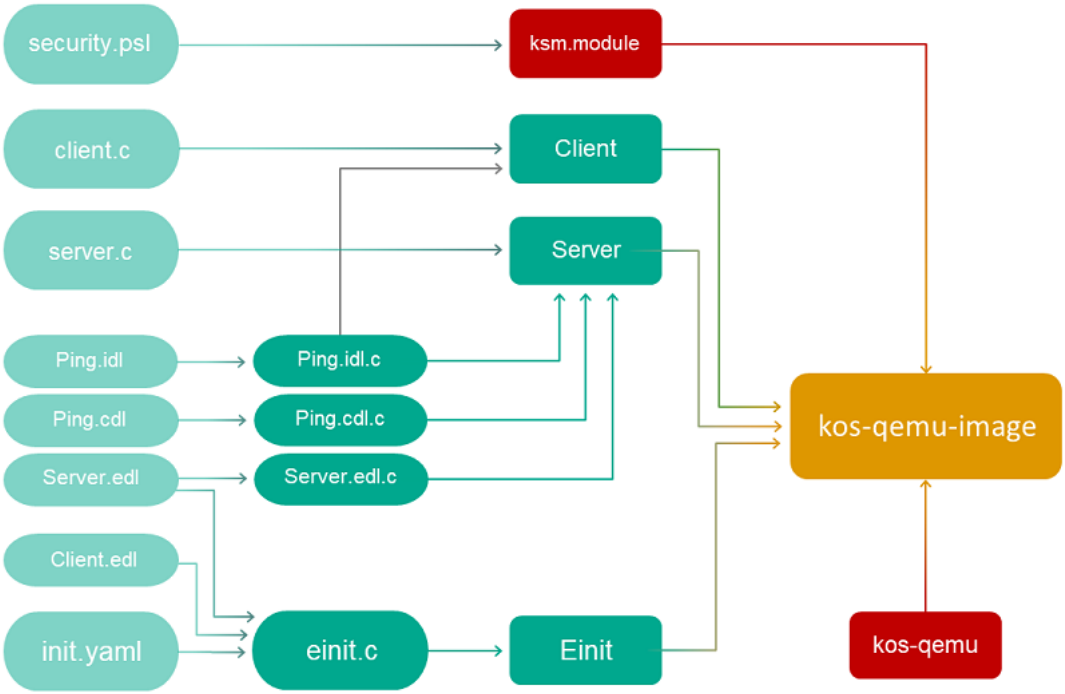
For the basic `hello` example included in KasperskyOS Community Edition that contains one application that does not provide any services, the build scenario looks as follows:



Example 2

The `echo` example included in KasperskyOS Community Edition describes a basic case of interaction between two programs via an IPC mechanism. To set up this interaction, you will need to implement an interface with the `Ping` method on a server and put the `Ping` service into a new component (for example, `Responder`), and an instance of this component needs to be put into the EDL description of the `Server` program.

If a solution contains programs that utilize an IPC mechanism, the build scenario looks as follows:



Using CMake from the contents of KasperskyOS Community Edition

To automate the process of preparing the solution image, you need to configure the `CMake` build system. You can base this system on the build system parameters used in the examples from KasperskyOS Community Edition.

`CMakeLists.txt` files use the standard `CMake` syntax, and commands and macros from libraries provided in KasperskyOS Community Edition.

Recommended structure of project directories

When creating a KasperskyOS-based solution, it is recommended to use the following directory structure in a project:

- In the project root, create a CMakeLists.txt root file containing the general build instructions for the entire solution.
- The source code of each program being developed should be placed into a separate directory within the `src` subdirectory.
- Create CMakeLists.txt files for building each application in the corresponding directories.
- To generate the source code of the `Einit` program, you should create a separate `einit` directory containing the `src` subdirectory in which you should put the init.yaml.in and security.psl.in templates.

  Any other files that need to be included in the solution image can also be put into this directory.
- Create a CMakeLists.txt file for building the `Einit` program in the `einit` directory.
- The files of EDL, CDL and IDL descriptions should be put into the `resources` directory in the project root. (An EDL description can be generated by using the `generate_edl_file()` command.)

```
Example structure of project directories
```

```
example$ tree
.
├── CMakeLists.txt
├── hello
│   ├── CMakeLists.txt
│   ├── src
│   │   ├── hello.c
├── einit
│   ├── CMakeLists.txt
│   ├── src
│   │   ├── init.yaml.in
│   │   ├── security.psl.in
├── resources
│   ├── Hello.idl
│   ├── Hello.cdl
│   ├── Hello.edl
```

Building a solution image

To build a solution image, you must use the `cmake` tool (the `toolchain/bin/cmake` executable file from KasperskyOS Community Edition).

Build script example:

build.sh

```bash
#!/bin/bash

# Script to be run in the project root.
# You can get information about the cmake tool run parameters
# via the shell command cmake --help, and from
# the official CMake documentation.

export SDK_PREFIX="/opt/KasperskyOS-Community-Edition-<version>"
export TARGET="aarch64-kos"
export BOARD="RPI4_BCM2711"
export CMAKE="$SDK_PREFIX/toolchain/bin/cmake"
export BUILD="build"

# Initialize the build system
$CMAKE \
-G "Unix Makefiles" \
-D CMAKE_BUILD_TYPE:STRING=Release \
 -D CMAKE_TOOLCHAIN_FILE=$SDK_PREFIX/toolchain/share/toolchain-$TARGET-clang.cmake \
 -D BOARD="$BOARD" \
-S . \
 -B $BUILD

# Build
# To build a solution image for QEMU, you must specify the target defined in the
# NAME parameter of the CMake command build_kos_qemu_image() in the CMakeLists.txt file
# for building the Einit program.
# To build a solution image for the hardware platform, you must specify the target
# defined in the NAME parameter of the CMake command build_kos_hw_image() in the
# CMakeLists.txt file for building the Einit program.
# To build an SD card image for the hardware platform, you must specify the target
# defined in the IMAGE_NAME parameter of the CMake command build_sd_image() in the
# CMakeLists.txt file for building the Einit program.
# To build a solution image for QEMU and start QEMU with this image, you must
# specify the sim target.
$CMAKE --build $BUILD --target sim
```

CMakeLists.txt root file

The `CMakeLists.txt` root file contains general build instructions for the entire solution.

The `CMakeLists.txt` root file must contain the following commands:

- `cmake_minimum_required (VERSION 3.25)` indicates the minimum supported version of `CMake`.

  For a KasperskyOS-based solution build, `CMake` version 3.25 or later is required.

  The required version of `CMake` is provided in KasperskyOS Community Edition and is used by default.

- `include (platform)` connects the `platform` library of `CMake`.

- `initialize_platform()` initializes the `platform` library.

- `project_header_default("STANDARD_GNU_17:YES" "STRICT_WARNINGS:NO")` sets the flags of the compiler and linker.

- **[Optional]** Connect and configure packages for the provided system programs and drivers that need to be included in the solution:

  - A package is connected by using the `find_package()` command.

  - After connecting a package, you must add the package-related directories to the list of search directories by using the `include_directories()` command.

  - For some packages, you must also set the values of properties by using the `set_target_properties()` command.

  > `CMake` descriptions of system programs and drivers provided in KasperskyOS Community Edition, and descriptions of their exported variables and properties are located in the corresponding files at `/opt/KasperskyOS-Community-Edition-<version>/sysroot-*-kos/lib/cmake/<program name>/<program name>-config.cmake`

- The `Einit` initializing program must be built using the `add_subdirectory(einit)` command.

- All applications to be built must be added by using the `add_subdirectory(<program directory name>)` command.

Example CMakeLists.txt root file

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.25)
project (example)

# Initializes the CMake library for the KasperskyOS SDK.
include (platform)
initialize_platform ()

# Set the linker and compiler flags.
project_header_default ("STANDARD_GNU_17:YES" "STRICT_WARNINGS:NO")

# Add package importing components for working with Virtual File System.
# Components are imported from the following file: /opt/KasperskyOS-Community-Edition-<version>/sysroot-*-
kos/lib/cmake/vfs/vfs-config.cmake
find_package (vfs REQUIRED COMPONENTS ENTITY CLIENT_LIB)
include_directories (${vfs_INCLUDE})

# Build the Hello application.
add_subdirectory (hello)

# Build the Einit initializing program.
add_subdirectory (einit)
```

CMakeLists.txt files for building applications

The `CMakeLists.txt` file for building an application must contain the following commands:

- `include (platform/nk)` connects the `CMake` library for working with the NK compiler.
- `project_header_default ("STANDARD_GNU_17:YES" "STRICT_WARNINGS:NO")` sets the flags of the compiler and linker.
- An EDL description of a process class for a program can be generated by using the `generate_edl_file()` command.
- If the program provides endpoints using an IPC mechanism, the following transport code must be generated:

  a. idl.h files are generated by the `nk_build_idl_files()` command

  b. cdl.h files are generated by the `nk_build_cdl_files()` command

  c. edl.h files are generated by the `nk_build_edl_files()` command
- `add_executable (<program name> "<path to the file containing the program source code>")` adds the program build target.
- `add_dependencies (<program name> <name of the edl.h file build target>)` adds a program build dependency on edl.h file generation.
- `target_link_libraries (<program name> <list of libraries>)` determines the libraries that need to be linked with the program during the build.

  For example, if the program uses file I/O or network I/O, it must be linked with the `vfs::client` transport library.

  > `CMake` descriptions of system programs and drivers provided in KasperskyOS Community Edition, and descriptions of their exported variables and properties are located in the corresponding files at `/opt/KasperskyOS-Community-Edition-<version>/sysroot-*-kos/lib/cmake/<program name>/<program name>-config.cmake`

- To automatically add descriptions of IPC channels to the `init.yaml` file when building a solution, you must define the `EXTRA_CONNECTIONS` property and assign it a value with descriptions of the relevant IPC channels.

  Please note the indentations at the beginning of strings in the `EXTRA_CONNECTIONS` property. These indentations are necessary to correctly insert values into the `init.yaml` file and must comply with its syntax requirements.

  Example of creating an IPC channel between a `Client` process and a `Server` process:

  ```
  set_target_properties (Client PROPERTIES
  EXTRA_CONNECTIONS
  "   - target: Server
        id: server_connection")
  ```

  When building this solution, the description of this IPC channel will be automatically added to the `init.yaml` file when processing macros of the init.yaml.in template.

- To automatically add a list of arguments for the `main()` function and a dictionary of environment variables to the `init.yaml` file when building a solution, you must define the `EXTRA_ARGS` and `EXTRA_ENV` properties and assign the appropriate values to them.

  Note the indentations at the beginning of strings in the `EXTRA_ARGS` and `EXTRA_ENV` properties. These indentations are necessary to correctly insert values into the `init.yaml` file and must comply with its syntax requirements.

  Example of sending the `Client` program the `"-v"` argument of the `main()` function and the environment variable `VAR1` set to `VALUE1`:

```
set_target_properties (Client PROPERTIES
EXTRA_ARGS
"    - \"-v\""
EXTRA_ENV
"     VAR1: VALUE1")
```

  When building this solution, the description of the `main()` function argument and the environment variable value will be automatically added to the `init.yaml` file when processing macros of the init.yaml.in template.

Example CMakeLists.txt file for building a simple application

CMakeLists.txt

```
project (hello)

# Include the CMake library named nk for working with the NK compiler (nk-gen-c).
include (platform/nk)

# Set the linker and compiler flags.
project_header_default ("STANDARD_GNU_17:YES" "STRICT_WARNINGS:NO")

# Define the name of the project that includes the program.
set (LOCAL_MODULE_NAME "example")

# Define the program name.
set (TASK_NAME "Hello")
# Please note the contents of the init.yaml.in and security.psl.in templates
# They define program names as ${LOCAL_MODULE_NAME}.${TASK_NAME}

# Define the targets that will be used to create the generated files of the program.
set (TASK_IDL_TARGET ${TASK_NAME}_idl)
set (TASK_CDL_TARGET ${TASK_NAME}_cdl)
set (TASK_EDL_TARGET ${TASK_NAME}_edl)

# Add the idl.h file build target.
nk_build_idl_files (${TASK_IDL_TARGET}
                    NK_MODULE ${LOCAL_MODULE_NAME}
                    IDL "../resources/Hello.idl"
                    )

# Add the cdl.h file build target.
nk_build_cdl_files (${TASK_CDL_TARGET}
                    IDL_TARGET ${TASK_IDL_TARGET}
                    NK_MODULE ${LOCAL_MODULE_NAME}
                    CDL "../resources/Hello.cdl"
                    )

# Add the EDL file build target.The EDL_FILE variable is exported
# and contains the path to the generated EDL file.
generate_edl_file (${TASK_NAME}
                   PREFIX ${LOCAL_MODULE_NAME}
                   )

# Add the edl.h file build target.
nk_build_edl_files (${TASK_EDL_TARGET}
                    NK_MODULE ${LOCAL_MODULE_NAME}
                    EDL ${EDL_FILE}
                    )

# Define the target for the program build.
add_executable (${TASK_NAME} "src/hello.c")
```

```
# Libraries that are linked to the program during the build.
target_link_libraries (${TASK_NAME}
                       PUBLIC vfs::client   # The program uses file I/O
                                            # and must be connected as a client to VFS
                       )

# Add a dependency on the Hello_edl target to the Hello target.
# The edl.h file must be generated before the Hello target is built.
add_dependencies (${TASK_NAME} ${TASK_EDL_TARGET})
```

CMakeLists.txt file for building the Einit program

The `CMakeLists.txt` file for building the `Einit` initializing program must contain the following commands:

- `include (platform/image)` connects the `CMake` library that contains the solution image build scripts.
- `project_header_default ("STANDARD_GNU_17:YES" "STRICT_WARNINGS:NO")` sets the flags of the compiler and linker.
- Configure the packages of system programs and drivers that need to be included in the solution.
  - A package is connected by using the `find_package ()` command.
  - For some packages, you must also set the values of properties by using the `set_target_properties ()` command.

  > `CMake` descriptions of system programs and drivers provided in KasperskyOS Community Edition, and descriptions of their exported variables and properties are located in the corresponding files at `/opt/KasperskyOS-Community-Edition-<version>/sysroot-*-kos/lib/cmake/<program name>/<program name>-config.cmake`

- To automatically add descriptions of IPC channels between processes of system programs to the `init.yaml` file when building a solution, you must add these channels to the `EXTRA_CONNECTIONS` property for the corresponding programs.

  Please note the indentations at the beginning of strings in the `EXTRA_CONNECTIONS` property. These indentations are necessary to correctly insert values into the `init.yaml` file and must comply with its syntax requirements.

  For example, the `VFS` program does not have a channel for connecting to the `Env` program by default. To automatically add a description of this channel to the init.yaml file during a solution build, you must add the following call to the `CMakeLists.txt` file for building the `Einit` program:

  ```
  set_target_properties (vfs_entity::entity PROPERTIES
  EXTRA_CONNECTIONS
  "  - target: env.Env
       id: {var: ENV_SERVICE_NAME, include: env/env.h}"
  ```

  When building this solution, the description of this IPC channel will be automatically added to the `init.yaml` file when processing macros of the init.yaml.in template.

- To automatically add a list of arguments for the `main()` function and a dictionary of environment variables to the `init.yaml` file when building a solution, you must define the `EXTRA_ARGS` and `EXTRA_ENV` properties and assign the appropriate values to them.

Note the indentations at the beginning of strings in the `EXTRA_ARGS` and `EXTRA_ENV` properties. These indentations are necessary to correctly insert values into the `init.yaml` file and must comply with its syntax requirements.

Example of sending the `VfsEntity` program the `"-f fstab"` argument of the `main()` function and the environment variable `ROOTFS` set to `ramdisk0,0 / ext2 0`:

```
set_target_properties (vfs_entity::entity PROPERTIES
EXTRA_ARGS
"    - \"-f\"
     - \"fstab\""
EXTRA_ENV
"     ROOTFS: ramdisk0,0 / ext2 0")
```

When building this solution, the description of the `main()` function argument and the environment variable value will be automatically added to the `init.yaml` file when processing macros of the init.yaml.in template.

- `set(ENTITIES <full list of programs included in the solution>)` defines the `ENTITIES` variable containing a list of executable files of all programs included in the solution.
- One or more commands for building the solution image:
  - build_kos_hw_image() creates the target for building a solution image for the hardware platform.
  - build_sd_image() – creates the target for building an SD card image for running a solution on the hardware platform.
  - build_kos_qemu_image() creates the target for building a solution image for QEMU.

Example CMakeLists.txt file for building the Einit program

CMakeLists.txt

```
project (einit)

# Connect the library containing solution image build scripts.
include (platform/image)
include (${KL_SDK_ROOT_PATH}/common/build-sd-image.cmake)

# Set the linker and compiler flags.
project_header_default ("STANDARD_GNU_17:YES" "STRICT_WARNINGS:NO")

# Define the CONNECTIONS_CFG_FILE variable containing the path to the init.yaml.in template.
set (CONNECTIONS_CFG_FILE "src/init.yaml.in")
# Define the SECURITY_PSL variable containing the path to the security.psl.in template.
set (SECURITY_PSL "src/security.psl.in")

# Configure the VFS program.
# By default, the VFS program is not mapped to a program implementing a block device.
# If you need to use a block device, such as ata from the ata component,
# you must define this device in the blkdev::entity_REPLACEMENT variable
# For more information about exported variables and properties of the VFS program,
# see /opt/KasperskyOS-Community-Edition-<version>/sysroot-*-kos/lib/cmake/vfs/vfs-config.cmake
# find_package(ata)
# set_target_properties (vfs_entity::entity PROPERTIES blkdev::entity_REPLACEMENT ata::entity)
# In the simplest case, you do not need to interact with a drive.
# For this reason, we set the value of the blkdev::entity_REPLACEMENT variable equal to an empty string
set_target_properties (vfs_entity::entity PROPERTIES blkdev::entity_REPLACEMENT "")

# Define the ENTITIES variable with a list of executable files of programs.
# It is important to include all programs that are part of the project, except the Einit program.
# Please note that the name of the executable file of a program must
# match the name of the target indicated in add_executable() in the CMakeLists.txt file for building this program.
set(ENTITIES
    vfs_entity::entity
    Hello
    )
```

```
# Create the build target named kos-image, which is a solution image for the hardware platform.
build_kos_hw_image (kos-image
                    EINIT_ENTITY EinitHw
                    CONNECTIONS_CFG ${CONNECTIONS_CFG_FILE}
                    SECURITY_PSL ${SECURITY_PSL}
                    IMAGE_FILES ${ENTITIES}
                    )

# Create a build target named sd-image that can be used to create an SD-card image
# for running a solution on the hardware platform.
build_sd_image (sd-image
                KOS_IMAGE_TARGET kos-image
                )


# Create the build target named kos-qemu-image, which is a solution image for QEMU.
build_kos_qemu_image (kos-qemu-image
                      EINIT_ENTITY EinitQemu
                      CONNECTIONS_CFG ${CONNECTIONS_CFG_FILE}
                      SECURITY_PSL ${SECURITY_PSL}
                      IMAGE_FILES ${ENTITIES}
                      )
```

init.yaml.in template

The `init.yaml.in` template is used to automatically generate a *part* of the `init.yaml` file prior to building the `Einit` program using `CMake` tools.

When using the `init.yaml.in` template, you do not have to manually add descriptions of system programs and the IPC channels for connecting to them to the `init.yaml` file.

The `init.yaml.in` template must contain the following data:

- Root `entities` key.
- List of all applications included in the solution.
- For applications that use an IPC mechanism, you must specify a list of IPC channels that connect this application to other applications.

  The IPC channels that connect this application to other *applications* are either indicated manually or specified in the CMakeLists.txt file for this application using the `EXTRA_CONNECTIONS` property.

  To specify a list of IPC channels that connect this application to system programs that are included in KasperskyOS Community Edition, the following macros are used:

  - `@INIT_<program name>_ENTITY_CONNECTIONS@` – during the build, this is replaced with the list of IPC channels containing all system programs that are linked to the application. The `target` and `id` fields are filled according to the `connect.yaml` files from KasperskyOS Community Edition located in `/opt/KasperskyOS-Community-Edition-<version>/sysroot-*-kos/include/<system program name>`).

    This macro needs to be used if the application does not have connections to other applications but instead connects *only to system programs*. This macro *adds* the root `connections` key.

  - `@INIT_<program name>_ENTITY_CONNECTIONS+@` – during the build, the list of IPC channels containing all system programs that are linked to the application is added to the manually defined list of IPC channels. This macro *does not add* the root `connections` key.

    This macro needs to be used if the application has connections to other applications that were manually indicated in the `init.yaml.in` template.

- The `@INIT_<program name>_ENTITY_CONNECTIONS@` and `@INIT_<program name>_ENTITY_CONNECTIONS+@` macros also add the list of connections for each program defined in the `EXTRA_CONNECTIONS` property when building this program.

- If you need to pass `main()` function arguments defined in the `EXTRA_ARGS` property to a program when building this program, you need to use the following macros:

  - `@INIT_<program name>_ENTITY_ARGS@` – during the build, this is replaced with the list of arguments of the `main()` function defined in the `EXTRA_ARGS` property. This macro *adds* the root `args` key.

  - `@INIT_<program name>_ENTITY_ARGS+@` – during the build, this macro adds the list of `main()` function arguments defined in the `EXTRA_ARGS` property to the list of manually defined arguments. This macro *does not add* the root `args` key.

- If you need to pass the values of environment variables defined in the `EXTRA_ENV` property to a program when building this program, you need to use the following macros:

  - `@INIT_<program name>_ENTITY_ENV@` – during the build, this is replaced with the dictionary of environment variables and their values defined in the `EXTRA_ENV` property. This macro *adds* the root `env` key.

  - `@INIT_<program name>_ENTITY_ENV+@` – during the build, this macro adds the dictionary of environment variables and their values defined in the `EXTRA_ENV` property to the manually defined variables. This macro *does not add* the root `env` key.

- `@INIT_EXTERNAL_ENTITIES@` – during the build, this macro is replaced with the list of system programs linked to the application and their IPC channels, `main()` function arguments, and values of environment variables.

Example init.yaml.in template

init.yaml.in

```
entities:

- name: ping.Client
  connections:
  # The "Client" program can query the "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@
```

When building the `Einit` program from this template, the following init.yaml file will be generated:

init.yaml

```
entities:

- name: ping.Client
  connections:
 # The "Client" program can query the "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
```

security.psl.in template

The `security.psl.in` template is used to automatically generate a *part* of the `security.psl` file prior to building the `Einit` program using `CMake` tools.

The `security.psl` file contains part of the solution security policy description.

When using the `security.psl.in` template, you do not have to manually add EDL descriptions of system programs to the `security.psl` file.

The `security.psl.in` template must contain a manually created solution security policy description, including the following declarations:

- Set the global parameters of a solution security policy
- Include PSL files in a solution security policy description
- Include EDL files of application software in a solution security policy description
- Create security model objects
- Bind methods of security models to security events
- Create security audit profiles

To automatically include system programs, the `@INIT_EXTERNAL_ENTITIES@` macro must be used.

Example security.psl.in template

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@

/* Startup of programs is allowed */
execute {
    grant ()
}
```

```
/* Sending and receiving requests, responses and errors is allowed. */
request {
    grant ()
}
response {
    grant ()
}

error {
    grant ()
}
/* Queries via the security interface are ignored. */
security {
    grant ()
}
```

CMake libraries in KasperskyOS Community Edition

This section contains a description of the libraries that are provided in KasperskyOS Community Edition for automatically building a KasperskyOS-based solution.

platform library

The `platform` library contains the following commands:

- `initialize_platform()` is the command for initializing the `platform` library.

  The `initialize_platform()` command can be called with the `FORCE_STATIC` parameter, which enables forced static linking of executable files:

  - By default, if the toolchain in the KasperskyOS SDK supports dynamic linking, the `initialize_platform()` command causes the `-rdynamic` flag to be used automatically for building all executable files defined via `CMake` `add_executable()` commands.

  - When calling `initialize_platform (FORCE_STATIC)` in the `CMakeLists.txt` root file, the toolchain supporting dynamic linking performs static linking of executable files.

  The `initialize_platform()` command can be called with the `NO_NEW_VERSION_CHECK` parameter, which disables the check for SDK updates and transmission of the SDK version to the Kaspersky server.

  To disable the check for SDK updates and transmission of SDK version data to the Kaspersky server, use the following call during the solution build: `initialize_platform(NO_NEW_VERSION_CHECK)`. For more details about the data provision policy, see Data provision.

- `project_static_executable_header_default()` is the command for enabling forced static linking of executable files defined via subsequent `CMake` `add_executable()` commands in one `CMakeLists.txt` file. The toolchain that supports dynamic linking performs static linking of these executable files.

- `platform_target_force_static()` is the command for enabling forced static linking of an executable file defined via the `CMake` `add_executable()` command. The toolchain that supports dynamic linking performs static linking of this executable file. For example, if the `CMake` commands `add_executable(client "src/client.c")` and `platform_target_force_static(client)` are called, static linking is performed for the `client` program.

- `project_header_default()` is the command for setting compile flags.

Command parameters are defined in pairs consisting of a compile flag and its value: `"FLAG_1:VALUE_1" "FLAG_2:VALUE_2" ... "FLAG_N:VALUE_N"`. The `CMake platform` library converts these pairs into compiler parameters.

When using the `initialize_platform(FORCE_STATIC)`, `project_static_executable_header_default()` and `platform_target_force_static()` commands, you may encounter linking errors if the static variants of the required libraries are missing (for example, if they were not built or are not included in the KasperskyOS SDK). Even if the static variants of the required libraries are available, these errors may still occur because the build system searches for the dynamic variants of required libraries by default instead of the expected static variants when using the `initialize_platform(FORCE_STATIC)`, `project_static_executable_header_default()` and `platform_target_force_static()` commands. To avoid errors, first make sure that the static variants are available. Then configure the build system to search for static libraries (although this capability may not be available for some libraries), or explicitly define linking with static libraries.

Examples of configuring the build system to search for static libraries:

```
set (fmt_USE_STATIC ON)
find_package (fmt REQUIRED)

set (fdn_USE_STATIC ON)
find_package (fdn REQUIRED)

set (sqlite_wrapper_USE_STATIC ON)
find_package (sqlite_wrapper REQUIRED)
```

Example that explicitly defines linking with a static library:

```
target_link_libraries(${PROJECT_NAME} PUBLIC logger::logger-static)
```

For more details about using dynamic libraries, see Using dynamic libraries.

These commands are used in `CMakeLists.txt` files for the Einit program and application software.

nk library

This section contains a description of the commands and macros of the `CMake` library for working with the NK compiler.

generate_edl_file()

This command is declared in the file `/opt/KasperskyOS-Community-Edition-<version>/toolchain/share/cmake/Modules/platform/nk2.cmake`.

```
generate_edl_file(NAME ...)
```

This command generates an EDL file containing a description of the process class.

Parameters:

- `NAME` – name of the EDL file being created. Required parameter.

- `PREFIX` – parameter in which you need to specify the name of the process class, excluding the name of the EDL file. For example, if the name of the process class for which the EDL file is being created is defined as `kl.core.NameServer`, the `PREFIX` parameter must pass the value `kl.core`.

- `EDL_COMPONENTS` – name of the component and its instance that will be included in the EDL file. For example: `EDL_COMPONENTS "env: kl.Env"`. To include multiple components, you need to use multiple `EDL_COMPONENTS` parameters.

- `SECURITY` – qualified name of the security interface method that will be included in the EDL file.

- `OUTPUT_DIR` – directory in which the EDL file will be created. The default directory is `${CMAKE_CURRENT_BINARY_DIR}`.

As a result of this command, the `EDL_FILE` variable is exported and contains the path to the generated EDL file.

Example call:

```
generate_edl_file(${ENTITY_NAME} EDL_COMPONENTS "env: kl.Env")
```

For an example of using this command, see the article titled CMakeLists.txt files for building application software.

nk_build_idl_files()

This command is declared in the file `/opt/KasperskyOS-Community-Edition-<version>/toolchain/share/cmake/Modules/platform/nk2.cmake`.

```
nk_build_idl_files(NAME ...)
```

This command creates a `CMake` target for generating `.idl.h` files for one or more defined IDL files using the NK compiler.

Parameters:

- `NAME` – name of the `CMake` target for building `.idl.h` files. If a target has not yet been created, it will be created by using `add_library()` with the specified name. Required parameter.

- `NOINSTALL` – if this option is specified, files will only be generated in the working directory and will not be installed in global directories: `${CMAKE_BINARY_DIR}/_headers_` `${CMAKE_BINARY_DIR}/_headers_/${PROJECT_NAME}`.

- `NK_MODULE` – parameter in which you need to specify the package name, excluding the name of the IDL file. For example, if the package name in the IDL description is defined as `kl.core.NameServer`, the `kl.core` value must be passed in the `NK_MODULE` parameter.

- `WORKING_DIRECTORY` – working directory for calling the NK compiler, which is `${CMAKE_CURRENT_BINARY_DIR}` by default.

- `DEPENDS` – additional build targets on which the IDL file depends.

  To add multiple targets, you need to use multiple `DEPENDS` parameters.

- `IDL` – path to the IDL file for which the idl.h file is being generated. Required parameter.

  To add multiple IDL files, you need to use multiple `IDL` parameters.

> If one IDL file imports another IDL file, idl.h files need to be generated in the order necessary for compliance with dependencies (with the most deeply nested first).

- `NK_FLAGS` – additional flags for the NK compiler.

Example call:

```
nk_build_idl_files (echo_idl_files NK_MODULE "echo" IDL "resources/Ping.idl"
                    NK_FLAGS "--extended-errors" "--trace-client-ipc=headers")
```

For an example of using this command, see the article titled CMakeLists.txt files for building application software.

nk_build_cdl_files()

This command is declared in the file `/opt/KasperskyOS-Community-Edition-<version>/toolchain/share/cmake/Modules/platform/nk2.cmake`.

```
nk_build_cdl_files(NAME ...)
```

This command creates a `CMake` target for generating `.cdl.h` files for one or more defined CDL files using the NK compiler.

Parameters:

- `NAME` – name of the `CMake` target for building `.cdl.h` files. If a target has not yet been created, it will be created by using `add_library()` with the specified name. Required parameter.

- `NOINSTALL` – if this option is specified, files will only be generated in the working directory and are not installed in global directories: `${CMAKE_BINARY_DIR}/_headers_` `${CMAKE_BINARY_DIR}/_headers_/${PROJECT_NAME}`.

- `IDL_TARGET` – target when building `.idl.h` files for IDL files containing descriptions of endpoints provided by components described in CDL files.

- `NK_MODULE` – parameter in which you need to specify the component name, excluding the name of the CDL file. For example, if the component name in the CDL description is defined as `kl.core.NameServer`, the `kl.core` value must be passed in the `NK_MODULE` parameter.

- `WORKING_DIRECTORY` – working directory for calling the NK compiler, which is `${CMAKE_CURRENT_BINARY_DIR}` by default.

- `DEPENDS` – additional build targets on which the CDL file depends.

  To add multiple targets, you need to use multiple `DEPENDS` parameters.

- `CDL` – path to the CDL file for which the `.cdl.h` file is being generated. Required parameter.

  To add multiple CDL files, you need to use multiple `CDL` parameters.

- `NK_FLAGS` – additional flags for the NK compiler.

Example call:

```
nk_build_cdl_files (echo_cdl_files IDL_TARGET echo_idl_files NK_MODULE "echo" CDL "resources/Ping.cdl")
```

For an example of using this command, see the article titled CMakeLists.txt files for building application software.

nk_build_edl_files()

This command is declared in the file `/opt/KasperskyOS-Community-Edition-<version>/toolchain/share/cmake/Modules/platform/nk2.cmake`.

```
nk_build_edl_files(NAME ...)
```

This command creates a `CMake` target for generating an `.edl.h` file for one defined EDL file using the NK compiler.

Parameters:

- `NAME` – name of the `CMake` target for building an `.edl.h` file. If a target has not yet been created, it will be created by using `add_library()` with the specified name. Required parameter.
- `NOINSTALL` – if this option is specified, files will only be generated in the working directory and are not installed in global directories: `${CMAKE_BINARY_DIR}/_headers_` `${CMAKE_BINARY_DIR}/_headers_/${PROJECT_NAME}`.
- `CDL_TARGET` – target when building `.cdl.h` files for CDL files containing descriptions of components of the EDL file for which the build is being performed.
- `IDL_TARGET` – target when building .idl.h files for IDL files containing descriptions of interfaces of the EDL file for which the build is being performed.
- `NK_MODULE` – parameter in which you need to specify the name of the process class, excluding the name of the EDL file. For example, if the process class name in the EDL description is defined as `kl.core.NameServer`, the `kl.core` value must be passed in the `NK_MODULE` parameter.
- `WORKING_DIRECTORY` – working directory for calling the NK compiler, which is `${CMAKE_CURRENT_BINARY_DIR}` by default.
- `DEPENDS` – additional build targets on which the EDL file depends.

  To add multiple targets, you need to use multiple `DEPENDS` parameters.
- `EDL` – path to the EDL file for which the edl.h file is being generated. Required parameter.
- `NK_FLAGS` – additional flags for the NK compiler.

Example calls:

```
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" NK_FLAGS "--extended-errors")
```

For an example of using this command, see the article titled CMakeLists.txt files for building application software.

Generating transport code for development in C++

The `CMake` commands add_nk_idl(), add_nk_cdl() and add_nk_edl() are used to generate transport proxy objects and stubs using the `nkppmeta` compiler when building a solution.

add_nk_idl()

This command is declared in the file `/opt/KasperskyOS-Community-Edition-<version>/toolchain/share/cmake/Modules/platform/nk2.cmake`.

```
add_nk_idl(NAME IDL_FILE ...)
```

This command creates a `CMake` target for generating a `*.idl.cpp.h` header file for a defined IDL file using the `nkppmeta` compiler. The command also creates a library containing the transport code for the defined interface. To link to this library, use the `bind_nk_targets()` command.

The generated header files contain a C++ representation for the interface and data types described in the IDL file, and the methods required for use of proxy objects and stubs.

Parameters:

- `NAME` – name of the `CMake` target. Required parameter.
- `IDL_FILE` – path to the IDL file. Required parameter.
- `NK_MODULE` – parameter in which you need to specify the package name, excluding the name of the IDL file. For example, if the package name in the IDL description is defined as `kl.core.NameServer`, the `kl.core` value must be passed in the `NK_MODULE` parameter.
- `LANG` – parameter in which you need to specify the `CXX` value.

Example call:

```
add_nk_idl (ANIMAL_IDL "${CMAKE_SOURCE_DIR}/resources/Animal.idl"
            NK_MODULE  "example"
            LANG       "CXX")
```

add_nk_cdl()

This command is declared in the file `/opt/KasperskyOS-Community-Edition-<version>/toolchain/share/cmake/Modules/platform/nk2.cmake`.

```
add_nk_cdl(NAME CDL_FILE ...)
```

This command creates a `CMake` target for generating a `*.cdl.cpp.h` file for a defined CDL file using the `nkppmeta` compiler. The command also creates a library containing the transport code for the defined component. To link to this library, use the `bind_nk_targets()` command.

The `*.cdl.cpp.h` file contains the tree of embedded components and endpoints provided by the component described in the CDL file.

Parameters:

- `NAME` – name of the `CMake` target. Required parameter.
- `CDL_FILE` – path to the CDL file. Required parameter.
- `NK_MODULE` – parameter in which you need to specify the component name, excluding the name of the CDL file. For example, if the component name in the CDL description is defined as `kl.core.NameServer`, the `kl.core` value must be passed in the `NK_MODULE` parameter.
- `LANG` – parameter in which you need to specify the `CXX` value.

Example call:

```
add_nk_cdl (CAT_CDL    "${CMAKE_SOURCE_DIR}/resources/Cat.cdl"
            NK_MODULE "example"
            LANG      "CXX")
```

add_nk_edl()

This command is declared in the file `/opt/KasperskyOS-Community-Edition-<version>/toolchain/share/cmake/Modules/platform/nk2.cmake`.

```
add_nk_edl(NAME EDL_FILE ...)
```

This command creates a `CMake` target for generating a `*.edl.cpp.h` file for a defined EDL file using the `nkppmeta` compiler. The command also creates a library containing the transport code for the server or client program. To link to this library, use the `bind_nk_targets()` command.

The `*.edl.cpp.h` file contains the tree of embedded components and endpoints provided by the process class described in the EDL file.

Parameters:

- `NAME` – name of the `CMake` target. Required parameter.
- `EDL_FILE` – path to the EDL file. Required parameter.
- `NK_MODULE` – parameter in which you need to specify the name of the process class, excluding the name of the EDL file. For example, if the process class name in the EDL description is defined as `kl.core.NameServer`, the `kl.core` value must be passed in the `NK_MODULE` parameter.
- `LANG` – parameter in which you need to specify the `CXX` value.

Example call:

```
add_nk_edl (SERVER_EDL "${CMAKE_SOURCE_DIR}/resources/Server.edl"
            NK_MODULE  "example"
            LANG       "CXX")
```

image library

This section contains a description of the commands and macros of the `CMake` library named `image` that is included in KasperskyOS Community Edition and contains solution image build scripts.

build_kos_qemu_image()

This command is declared in the file `/opt/KasperskyOS-Community-Edition-<version>/toolchain/share/cmake/Modules/platform/image.cmake`.

```
build_kos_qemu_image(NAME ...)
```

The command creates a `CMake` target for building a solution image for QEMU.

Parameters:

- `NAME` – name of the `CMake` target for building a solution image. Required parameter.
- `PERFCNT_KERNEL` – use the kernel with performance counters if it is available in KasperskyOS Community Edition.
- `EINIT_ENTITY` – name of the executable file that will be used to start the `Einit` program.
- `EXTRA_XDL_DIR` – additional directories to include when building the `Einit` program.
- `CONNECTIONS_CFG` – path to the `init.yaml` file or init.yaml.in template.
- `SECURITY_PSL` – path to the `security.psl` file or security.psl.in template.
- `KLOG_ENTITY` – target for building the `Klog` system program, which is responsible for the security audit. If the target is not specified, the audit is not performed.
- `QEMU_FLAGS` – additional flags for running QEMU.
- `IMAGE_BINARY_DIR_BIN` – directory for the final image and other artifacts. It matches `CMAKE_CURRENT_BINARY_DIR` by default.
- `NO_AUTO_BLOB_CONTAINER` – solution image will not include the `BlobContainer` program that is required for working with dynamic libraries in shared memory. For more details, refer to Including the BlobContainer system program in a KasperskyOS-based solution.
- `PACK_DEPS`, `PACK_DEPS_COPY_ONLY`, `PACK_DEPS_LIBS_PATH`, and `PACK_DEPS_COPY_TARGET` – parameters that define the method used to add dynamic libraries to the solution image.
- `GINGER_ENABLE` – enable (`GINGER_ENABLE TRUE`) or disable (`GINGER_ENABLE FALSE`) the Ginger template engine.
- `GINGER_DEFINITIONS` – set of variables used when expanding Ginger PSL templates. For example, `GINGER_DEFINITIONS "foo=bar baz=quux USE_DYNLD"` sets the `foo` variable to `bar`, sets the `baz`

variable to `quux`, and sets the `USE_DYNLD` variable to `TRUE`. The `USE_DYNLD` variable is set to `TRUE` because variables used when expanding Ginger PSL templates are set to `TRUE` by default.

- `GINGER_DUMP_DIR` – path to the directory where PSL files received from Ginger PSL templates will be stored. These PSL files are needed only to verify what was obtained as a result of expanding Ginger PSL templates, and are not used to generate source code of the Kaspersky Security Module. (When source code of the security module is generated, Ginger PSL templates are expanded in memory irrespective of whether or not the `GINGER_DUMP_DIR` parameter is being used.) The names of PSL files are generated based on the absolute paths to Ginger PSL templates. For example, the PSL file named `foo!bar!baz.psl` corresponds to the Ginger PSL template located at the path `/foo/bar/baz.psl`.

- `IMAGE_FILES` – executable files of applications and system programs (except the `Einit` program) and any other files to be added to the ROMFS image.

  To add multiple applications or files, you can use multiple `IMAGE_FILES` parameters.

- `<path to files>` – free parameters like `IMAGE_FILES`.

Example call:

```
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})
```

For an example of using this command, see the article titled CMakeLists.txt files for building the Einit program.

build_kos_hw_image()

This command is declared in the file `/opt/KasperskyOS-Community-Edition-<version>/toolchain/share/cmake/Modules/platform/image.cmake`.

```
build_kos_hw_image(NAME ...)
```

The command creates a `CMake` target for building a solution image for the hardware platform.

Parameters:

- `NAME` – name of the `CMake` target for building a solution image. Required parameter.

- `PERFCNT_KERNEL` – use the kernel with performance counters if it is available in KasperskyOS Community Edition.

- `EINIT_ENTITY` – name of the executable file that will be used to start the `Einit` program.

- `EXTRA_XDL_DIR` – additional directories to include when building the `Einit` program.

- `CONNECTIONS_CFG` – path to the init.yaml file or init.yaml.in template.

- `SECURITY_PSL` – path to the security.psl file or security.psl.in template.

- `KLOG_ENTITY` – target for building the `Klog` system program, which is responsible for the security audit. If the target is not specified, the audit is not performed.

- `IMAGE_BINARY_DIR_BIN` – directory for the final image and other artifacts. The default directory is `CMAKE_CURRENT_BINARY_DIR`.

- `NO_AUTO_BLOB_CONTAINER` – solution image will not include the `BlobContainer` program that is required for working with dynamic libraries in shared memory. For more details, refer to Including the BlobContainer system program in a KasperskyOS-based solution.

- `PACK_DEPS`, `PACK_DEPS_COPY_ONLY`, `PACK_DEPS_LIBS_PATH`, and `PACK_DEPS_COPY_TARGET` – parameters that define the method used to add dynamic libraries to the solution image.

- `GINGER_ENABLE` – enable (`GINGER_ENABLE TRUE`) or disable (`GINGER_ENABLE FALSE`) the Ginger template engine.

- `GINGER_DEFINITIONS` – set of variables used when expanding Ginger PSL templates. For example, `GINGER_DEFINITIONS "foo=bar baz=quux USE_DYNLD"` sets the `foo` variable to `bar`, sets the `baz` variable to `quux`, and sets the `USE_DYNLD` variable to `TRUE`. The `USE_DYNLD` variable is set to `TRUE` because variables used when expanding Ginger PSL templates are set to `TRUE` by default.

- `GINGER_DUMP_DIR` – path to the directory where PSL files received from Ginger PSL templates will be stored. These PSL files are needed only to verify what was obtained as a result of expanding Ginger PSL templates, and are not used to generate source code of the Kaspersky Security Module. (When source code of the security module is generated, Ginger PSL templates are expanded in memory irrespective of whether or not the `GINGER_DUMP_DIR` parameter is being used.) The names of PSL files are generated based on the absolute paths to Ginger PSL templates. For example, the PSL file named `foo!bar!baz.psl` corresponds to the Ginger PSL template located at the path `/foo/bar/baz.psl`.

- `IMAGE_FILES` – executable files of applications and system programs (except the `Einit` program) and any other files to be added to the ROMFS image.

  To add multiple applications or files, you can use multiple `IMAGE_FILES` parameters.

- `<path to files>` – free parameters like `IMAGE_FILES`.

Example call:

```
build_kos_hw_image ( kos-image
                     EINIT_ENTITY EinitHw
                     CONNECTIONS_CFG "src/init.yaml.in"
                     SECURITY_CFG "src/security.cfg.in"
                     IMAGE_FILES ${ENTITIES})
```

For an example of using this command, see the article titled CMakeLists.txt files for building the Einit program.

build_sd_image library

The `build_sd_image` library is included in the KasperskyOS SDK and contains the `build_sd_image()` command. This command is declared in the file named `/opt/KasperskyOS-Community-Edition-<version>/common/build-sd-image.cmake`.

```
build_sd_image(IMAGE_NAME ...)
```

The command creates a CMake target for building an SD card image for running a solution on the hardware platform.

Parameters:

- `IMAGE_NAME` – name of the CMake target for building an SD card image for running the solution on the hardware platform. Required parameter.
- `KOS_IMAGE_TARGET` – name of the CMake target for building a solution image for the hardware platform that will be added to the SD card image.
- `KOS_COPY_PATH` – path to the directory where the solution image for the hardware platform will be copied before it is added to the SD card image.
- `IMAGE_FS` – path to the root directory of the file system that will be used for the SD card image. Default value: `${CMAKE_BINARY_DIR}/hdd`.
- `OUTPUT_IMAGE_NAME` – name of the SD card image.
- `DISK_SIZE` – size of the created SD card image in megabytes. Default value: 1024 MB.
- `PARTITION_CMD` – set of parameters that will be used to create and configure partitions in the SD card image. Separate the parameters with a space.

Example call:

```
include (${KL_SDK_ROOT_PATH}/common/build-sd-image.cmake)

set (IMAGE_FS             "${CMAKE_BINARY_DIR}/hdd")
set (IMAGE_PART1_FS       "${IMAGE_FS}/part1")
set (IMAGE_PART2_FS       "${IMAGE_FS}/part2")
set (DISK_IMAGE           hdd.img)

build_sd_image (sd-image
                KOS_IMAGE_TARGET  kos-image
                KOS_COPY_PATH     ${IMAGE_PART1_FS}
                IMAGE_FS          ${IMAGE_FS}
                DISK_SIZE         512
                OUTPUT_IMAGE_NAME ${DISK_IMAGE}
                PARTITION_CMD    -p1 ${IMAGE_PART1_FS} -%1 50 -f1 fat32
                                 -p2 ${IMAGE_PART2_FS} -%2 50 -f2 ext4)
```

kpa library

This section contains a description of commands of the `CMake` library `kpa`, which is intended for building KPA packages.

add_kpa_package()

This command is declared in the file `/opt/KasperskyOS-Community-Edition-<version>/toolchain/share/cmake/Modules/platform/kpa.cmake`.

```
add_kpa_package(KPA_TARGET_NAME ...)
```

The command creates a `CMake` target for building the KPA package. When building this target, the build process automatically creates the KPA package manifest and the KPA package, which includes all components that were added to this target using the `CMake` command add_kpa_component().

Parameters:

- `KPA_TARGET_NAME` – name of the `CMake` target. Required parameter.
- `MANIFEST_V` – version of the KPA package manifest. Required parameter. It must be set to `2`.
- `VERSION` – version of the KPA package. The default value is `PROJECT_VERSION`.
- `ID` – ID of the KPA package. The default value is `KPA_TARGET_NAME`.
- `DEVELOPER_ID` – identifier of the developer. The default value is `"unspecified"`.
- `DONT_VERIFY`—if this parameter is defined, verification of the presence of all KPA package components specified in its manifest and the absence of unspecified components, and calculation of the checksums of KPA package components and their comparison with those specified in the KPA package manifest will not be performed.

Properties of the created `CMake` target:

- `KPA_OUTPUT_DIR` – directory in which the KPA package will be put.
- `KPA_OUTPUT_NAME` – name of the KPA file without the file extension. The default value is `${DEVELOPER_ID}.${ID}`.

Example call:

```
include (platform/kpa)

add_kpa_package (Hello.kpa ALL ID "hello" DEVELOPER_ID test_dev MANIFEST_V "2")

set_target_properties(Hello.kpa PROPERTIES
                      KPA_OUTPUT_NAME "hello")
```

add_kpa_component()

This command is declared in the file `/opt/KasperskyOS-Community-Edition-<version>/toolchain/share/cmake/Modules/platform/kpa.cmake`.

```
add_kpa_component(PACKAGE_NAME MODE ...)
```

The command adds the component to the specified `CMake` target for building the KPA package. This command can be called multiple times for the same KPA package build target. A KPA package build target is created by the add_kpa_package() command.

Parameters:

- `PACKAGE_NAME` – name of the `CMake` target for building the KPA package. Required parameter.
- `MODE` – type of component. Required parameter. The following values are possible:
  - `LIBRARY` – the component is a dynamic library. You must specify the full path to the file or the name of the `CMake` target. It is put into the directory `/<package_name>/lib` when the KPA package is installed.

- `FILES` – the component is a file. You must specify the full path to the file. When a KPA package is installed, it is put into the `/<package_name>/res` directory.
- `DIRECTORY` – the component is a directory containing files. You must specify the full path to the directory. When a KPA package is installed, it is put into the `/<package_name>/res` directory.
- `RUN_CONFIGURATION` – the component is a program run configuration. You must specify the following parameters:
  - `ID` – startup configuration ID that is unique within the KPA package. Required parameter.
  - `NAME` – startup configuration name. Required parameter.
  - `TYPE` – type of run configuration: `"gui"` – process with a graphical user interface or `"service"` – endpoint process. Required parameter.
  - `PATH` – full path to the executable file of the component after KPA package installation relative to the `/<package_name>` directory.
  - `EIID` – security class of the program. Required parameter.
  - `PRIMARY` – indicates whether the startup configuration is the primary one during program startup.
  - `AUTORUN` – indicates whether this configuration is started automatically.
  - `ARGS` – list of command-line arguments in the form of a string array for starting the program.
  - `ENV` – list of environment variables for starting the program.
- `RESOURCE` – used for adding arbitrary data to the KPA package. You must specify the following parameters:
  - Full path to the file or directory containing the data. Required parameter.
  - In the `TYPE` parameter, specify the data type: `"res"`, `"bin"`, `"lib"` or `"manifestLocale"`. Required parameter. For more details, refer to List of "components" objects.
  - (Optional) In the `DIRECTORY_PATH` parameter, specify the path to the directory where the data will be put after KPA package installation; the specified path should be relative to the `/<package_name>/res` directory.

Example calls:

```
include (platform/kpa)
# Add the program executable file
add_kpa_component(Hello.kpa RESOURCE ".build/aarch64-kos-debug/hello/Hello" TYPE "bin" DIRECTORY_PATH "bin/")
Adding program startup configuration
add_kpa_component(Hello.kpa RUN_CONFIGURATION "app" NAME "Hello app" TYPE "service" EIID "hello.Hello" PATH
"bin/Hello")
```

Building without CMake

This section contains a description of the scripts, tools, compilers and build templates provided in KasperskyOS Community Edition.

These tools can be used:

- In other build systems.
- To perform individual steps of the build.
- To analyze the build specifications and write a custom build system.

The general scenario for building a solution image is described in the article titled Build process overview.

Tools for building a solution

This section contains a description of the scripts, tools, compilers and build templates provided in KasperskyOS Community Edition.

Build scripts and tools

KasperskyOS Community Edition includes the following build scripts and tools:

- nk-gen-c

  The NK compiler (`nk-gen-c`) generates transport code based on the IDL, CDL, and EDL descriptions. Transport code is needed for generating, sending, receiving, and processing IPC messages.

- nk-psl-gen-c

  The `nk-psl-gen-c` compiler generates the C-language source code of the Kaspersky Security Module based on the solution security policy description and the IDL, CDL, and EDL descriptions. The `nk-psl-gen-c` compiler also generates the C-language source code of solution security policy tests based on solution security policy tests in PAL.

- einit

  The `einit` tool automates the creation of code for the `Einit` initializing program. This program is the first to start when KasperskyOS is loaded. Then it starts all other programs and creates IPC channels between them.

- makekss

  The `makekss` script creates the Kaspersky Security Module.

- makeimg

  The `makeimg` script creates the final boot image of the KasperskyOS-based solution with all programs to be started and the Kaspersky Security Module.

nk-gen-c

The NK compiler (`nk-gen-c`) generates transport code based on the IDL, CDL, and EDL descriptions.

The `nk-gen-c` compiler receives the IDL, CDL or EDL file and creates the following files:

- A `*.*dl.h` file containing the transport code.

- A `*.*dl.nk.d` file that lists the created `*.*dl.h` file's dependencies on the IDL and CDL files. The `*.*dl.nk.d` file is created for the build system.

Syntax of the shell command for starting the `nk-gen-c` compiler:

```
nk-gen-c -I <SYSROOT_INCLUDE_PATH> [-I <PATH>]... [-o <PATH>] [--types]
[--interface] [--endpoints] [--client] [--server] [--extended-errors]
[(--deprecated-no-extended-errors)] [--trace-client-ipc {headers|dump}]
[--trace-server-ipc {headers|dump}] [--ipc-trace-method-filter <METHOD 1>[,<METHOD 2>]...]
[-h|--help] [--version] <FILE>
```

Basic parameters:

- `FILE`

  Path to the IDL, CDL, or EDL file for which you need to generate transport code.

- `-I <SYSROOT_INCLUDE_PATH>`

  Path to the `sysroot-*-kos/include` directory from the KasperskyOS SDK.

- `-I <PATH>`

  These parameters must be used to define the paths to directories containing IDL and CDL files that are referenced by the file defined via the `FILE` parameter.

- `-o <PATH>`

  Path to an existing directory where the created files will be placed. If this parameter is not specified, the created files will be put into the current directory.

- `-h|--help`

  Prints the Help text.

- `--version`

  Prints the version of the `nk-gen-c` compiler.

- `--extended-errors`

  This parameter provides the capability to use interface methods with one or more error parameters of user-defined IDL types. (The client works with error parameters like it works with output parameters.)

  > If the `--extended-errors` parameter is not specified, you can use interface methods only with one `status` error parameter of the IDL type `UInt16` whose value is received by the client via return code of the interface method. This mechanism for sending error parameters is obsolete and will no longer be supported in the future, so you are advised to always specify the `--extended-errors` parameter.

- `--deprecated-no-extended-errors`

  This parameter is the default parameter and provides the capability to use interface methods only with one `status` error parameter of the IDL type `UInt16` whose value is received by the client via return code of the interface method. If the `--deprecated-no-extended-errors` parameter is not specified and the `--extended-errors` parameter is also not specified, you can also use this mechanism to send error parameters. However, in this case there will be no warning about the need to specify the `--extended-errors` parameter or the `--deprecated-no-extended-errors` parameter. (You need to specify one of these parameters because the `--deprecated-no-extended-errors` parameter will eventually cease to be the default parameter in the future, and the `nk-gen-c` compiler will end with an error if you don't explicitly define the mechanism for sending error parameters.)

Selective generation of transport code

To reduce the volume of generated transport code, you can use flags for selective generation of transport code. For example, you can use the `--server` flag for programs that implement endpoints, and use the `--client` flag for programs that utilize the endpoints.

> If no selective generation flag for transport code is specified, the `nk-gen-c` compiler generates transport code with all possible methods and types for the defined IDL, CDL, or EDL file.

Flags for selective generation of transport code for an IDL file:

- `--types`

  The transport code includes the types corresponding to the IDL types from the defined IDL file, and the types corresponding to this file's imported IDL types that are used in IDL types of the defined IDL file. However, the types corresponding to imported IDL constants and to the aliases of imported IDL types are not included in the `*.idl.h` file. To use the types corresponding to imported IDL constants and to the aliases of imported IDL types, you must separately generate the transport code for the IDL files from which you are importing.

- `--interface`

  The transport code corresponds to the `--types` flag, and includes the types of structures of the constant part of IPC requests and IPC responses for interface methods whose signatures are specified in the defined IDL file. In addition, the transport code contains constants indicating the sizes of IPC message arenas.

- `--client`

  The transport code corresponds to the `--interface` flag, and includes the proxy object type, proxy object initialization method, and the interface methods specified in the defined IDL file.

- `--server`

  The transport code corresponds to the `--interface` flag, and includes the types and dispatcher (dispatch method) used to process IPC requests corresponding to the interface methods specified in the defined IDL file.

Flags for selective generation of transport code for a CDL or EDL file:

- `--types`

  The transport code includes the types corresponding to the IDL types that are used in the method parameters of endpoints provided by the component (for the defined CDL file) or process class (for the defined EDL file).

- `--endpoints`

  The transport code corresponds to the `--types` flag, and includes the types of structures of the constant part of IPC requests and IPC responses for the methods of endpoints provided by the component (for the defined CDL file) or process class (for the defined EDL file). In addition, the transport code contains constants indicating the sizes of IPC message arenas.

- `--client`

  The transport code corresponds to the `--types` flag, and includes the types of structures of the constant part of IPC requests and IPC responses for the methods of endpoints provided by the component (for the defined CDL file) or process class (for the defined EDL file). In addition, the transport code contains constants indicating the sizes of IPC message arenas, and the proxy object types, proxy object initialization methods, and methods of endpoints provided by the component (for the defined CDL file) or process class (for the defined EDL file).

- `--server`

  The transport code corresponds to the `--types` flag, and includes the types and dispatchers (dispatch methods) used to process IPC requests corresponding to the endpoints provided by the component (for the defined CDL file) or process class (for the defined EDL file). In addition, the transport code contains constants indicating the sizes of IPC message arenas, and the stub types, stub initialization methods, and the types of structures of the constant part of IPC requests and IPC responses for the methods of endpoints provided by the component (for the defined CDL file) or process class (for the defined EDL file).

Printing diagnostic information about sending and receiving IPC messages

Transport code can generate diagnostic information about sending and receiving IPC messages and print this data via standard error. To generate transport code with these capabilities, use the following parameters:

- `--trace-client-ipc {headers|dump}`

  The code for printing diagnostic information is executed directly before the `Call()` system call is executed and immediately after it is executed. If the `headers` value is specified, the diagnostic information includes the endpoint method ID (MID), the Runtime Implementation Identifier (RIID), the size of the constant part of the IPC message (in bytes), the contents of the IPC message arena handle, the size of the IPC message arena (in bytes), and the size of the utilized part of the IPC message arena (in bytes). If the `dump` value is specified, the diagnostic information additionally includes the contents of the constant part and arena of the IPC message in hexadecimal format.

  When using this parameter, you must either specify the selective generation flag for transport code `--client` or refrain from specifying a selective generation flag for transport code.

- `--trace-server-ipc {headers|dump}`

  The code for printing diagnostic information is executed directly before calling the function that implements the interface method, and immediately after the completion of this function. In other words, it is executed when the dispatcher (dispatch method) is called in the interval between execution of the `Recv()` and `Reply()` system calls. If the `headers` value is specified, the diagnostic information includes the endpoint method ID (MID), the Runtime Implementation Identifier (RIID), the size of the constant part of the IPC message (in bytes), the contents of the IPC message arena handle, the size of the IPC message arena (in bytes), and the size of the utilized part of the IPC message arena (in bytes). If the `dump` value is specified, the diagnostic information additionally includes the contents of the constant part and arena of the IPC message in hexadecimal format.

  When using this parameter, you must either specify the selective generation flag for transport code `--server` or refrain from specifying a selective generation flag for transport code.

- `--ipc-trace-method-filter <METHOD 1>[,<METHOD 2>]...`

  Diagnostic information is printed if only the defined interface methods are called. For the `METHOD` value, you can use the interface method name or the construct `<package name>:<interface method name>`. The package name and interface method name are specified in the IDL file.

  If this parameter is not specified, diagnostic information is printed when any interface method is called.

  This parameter can be specified multiple times. For example, you can specify all required interface methods in one parameter or specify each required interface method in a separate parameter.

nk-psl-gen-c

The `nk-psl-gen-c` compiler generates the C-language source code of the Kaspersky Security Module based on the solution security policy description and the IDL, CDL, and EDL descriptions. This code is used by the makekss script.

The `nk-psl-gen-c` compiler can also generate the C-language source code of solution security policy tests based on solution security policy tests in PAL.

Syntax of the shell command for starting the `nk-psl-gen-c` compiler:

```
nk-psl-gen-c {-I|--include-dir} <SYSROOT_INCLUDE_DIR> [{-I|--include-dir} <DIR>]...
[{-o|--output} <FILE>] [--out-tests <FILE>] [{-t|--tests} <ARG>] [{-a|--audit} <FILE>]
[--enable-preprocessor] [--preprocessor-definition=<VAR_NAME>[=<VAR_VALUE>]]...
[--preprocessor-dump-dir <DIR>] [-h|--help] [--version] <INPUT>
```

Parameters:

- `INPUT`

  Path to the top-level file of the solution security policy description. This is normally the `security.psl` file.

- `{-I|--include-dir} <SYSROOT_INCLUDE_DIR>`

  Path to the `sysroot-*-kos/include` directory from the KasperskyOS SDK.

- `{-I|--include-dir} <DIR>`

  These parameters must be used to define the paths to directories containing IDL, CDL, and EDL files pertaining to the solution, and the paths to directories containing auxiliary files from the KasperskyOS SDK (`common`, `toolchain/include`).

- `{-o|--output} <FILE>`

  Path to the file that will save the source code of the Kaspersky Security Module and (optionally) the source code of solution security policy tests. The path must include existing directories.

- `--out-tests <FILE>`

  Path to the file that will save the source code of the solution security policy tests.

- `{-t|--tests} <ARG>`

  Defines whether the source code of solution security policy tests must be generated. `ARG` can take the following values:

  - `skip` – source code of tests is not generated. This value is used by default if the `{-t|--tests} <ARG>` parameter is not specified.

  - `generate` – source code of tests is generated. If the source code of tests is generated, you are advised to use the `--out-tests <FILE>` parameter. Otherwise, the source code of tests will be saved in the same file containing the source code of the Kaspersky Security Module, which may lead to errors during the build.

- `{-a|--audit} <FILE>`

  Path to the file that will save the C-language source code of the audit decoder.

- `--enable-preprocessor`

  Enables the Ginger template engine.

- `--preprocessor-definition=<VAR_NAME>[=<VAR_VALUE>]`

  These parameters must be used to define the variables that are utilized when expanding Ginger PSL templates. For example, `--preprocessor-definition=foo=bar` sets the `foo` variable to `bar`, and `--preprocessor-definition=USE_DYNLD` sets the `USE_DYNLD` variable to `TRUE`. The `USE_DYNLD` variable is set to `TRUE` because variables used when expanding Ginger PSL templates are set to `TRUE` by default.

- `--preprocessor-dump-dir <DIR>`

  Path to the directory where PSL files received from Ginger PSL templates will be stored. These PSL files are needed only to verify what was obtained as a result of expanding Ginger PSL templates, and are not used to generate source code of the Kaspersky Security Module. (When source code of the security module is generated, Ginger PSL templates are expanded in memory irrespective of whether or not the `--preprocessor-dump-dir <DIR>` parameter is being used.) The names of PSL files are generated based on the absolute paths to Ginger PSL templates. For example, the PSL file named `foo!bar!baz.psl` corresponds to the Ginger PSL template located at the path `/foo/bar/baz.psl`.

- `-h|--help`

  Prints the Help text.

- `--version`

  Prints the version of the `nk-psl-gen-c` compiler.

einit

The `einit` tool automates the creation of code for the Einit initializing program.

The `einit` tool receives the solution initialization description (the `init.yaml` file by default) and EDL, CDL and IDL descriptions, and creates a file containing the source code of the `Einit` initializing program. Then the `Einit` program must be built using the C-language cross compiler that is provided in KasperskyOS Community Edition.

Syntax for using the `einit` tool:

```
einit -I PATH -o PATH [--help] FILE
```

Parameters:

- `FILE`

  Path to the `init.yaml` file.

- `-I PATH`

  Path to the directory containing the auxiliary files (including EDL, CDL and IDL descriptions) required for generating the initializing program. By default, these files are located in the directory `/opt/KasperskyOS-Community-Edition-<version>/sysroot-*-kos/include`.

- `-o, --out-file PATH`

  Path to the created .c file containing the code of the initializing program.

- `-h, --help`

  Displays the Help text.

makekss

The `makekss` script creates the Kaspersky Security Module.

The script calls the nk-psl-gen-c compiler to generate the source code of the security module, then compiles the resulting code by calling the C compiler that is provided in KasperskyOS Community Edition.

The script creates the security module from the solution security policy description.

Syntax for using the `makekss` script:

```
makekss --target=ARCH --module=PATH --with-nk="PATH" --with-nktype="TYPE" --with-nkflags="FLAGS" [--output="PATH"]
[--help][--with-cc="PATH"][--with-cflags="FLAGS"] FILE
```

Parameters:

- `FILE`

  Path to the top-level file of the solution security policy description.

- `--target=ARCH`

  Processor architecture for which the build is intended.

- `--module=-lPATH`

  Path to the `ksm_kss` library. This key is passed to the C compiler for linking to this library.

- `--with-nk=PATH`

  Path to the `nk-psl-gen-c` compiler that will be used to generate the source code of the security module. By default, the compiler is located in `/opt/KasperskyOS-Community-Edition-<version>/toolchain/bin/nk-psl-gen-c`.

- `--with-nktype="TYPE"`

  Indicates the type of NK compiler that will be used. To use the `nk-psl-gen-c` compiler, indicate the `psl` type.

- `--with-nkflags="FLAGS"`

  Parameters used when calling the `nk-psl-gen-c` compiler.

  The `nk-psl-gen-c` compiler will require access to all EDL, CDL and IDL descriptions. To enable the `nk-psl-gen-c` compiler to find these descriptions, you need to pass the paths to these descriptions in the `--with-nkflags` parameter by using the `-I` switch of the `nk-psl-gen-c` compiler.

- `--output=PATH`

  Path to the created security module file.

- `--with-cc=PATH`

  Path to the C compiler that will be used to build the security module. The compiler provided in KasperskyOS Community Edition is used by default.

- `--with-cflags=FLAGS`

  Parameters used when calling the C compiler.

- `-h, --help`

  Displays the Help text.

makeimg

The `makeimg` script creates the final boot image of the KasperskyOS-based solution with all executable files of programs and the Kaspersky Security Module.

The script receives a list of files, including the executable files of all applications that need to be added to ROMFS of the loaded image, and creates the following files:

- Solution image
- Solution image without character tables (`.stripped`)
- Solution image with debug character tables (`.dbg.syms`)

Syntax for using the makeimg script:

```
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
```

Parameters:

- `FILES`

  List of paths to files, including the executable files of all applications that need to be added to ROMFS.

  The security module (`ksm.module`) must be explicitly specified, or else it will not be included in the solution image. The `Einit` application does not need to be indicated because it will be automatically included in the solution image.

- `--target=ARCH`

  Architecture for which the build is intended.

- `--sys-root=PATH`

  Path to the root directory sysroot. By default, this directory is located in `/opt/KasperskyOS-Community-Edition-<version>/sysroot-*-kos/`.

- `--with-toolchain=PATH`

  Path to the set of auxiliary tools required for the solution build. By default, these tools are located in `/opt/KasperskyOS-Community-Edition-<version>/toolchain/`.

- `--ldscript=PATH`

  Path to the linker script required for the solution build. By default, this script is located in `/opt/KasperskyOS-Community-Edition-<version>/libexec/aarch64-kos/`.

- `--img-src=PATH`

  Path to the precompiled KasperskyOS kernel. By default, the kernel is located in `/opt/KasperskyOS-Community-Edition-<version>/libexec/aarch64-kos/`.

- `--img-dst=PATH`

  Path to the created image file.

- `--with-init=PATH`

  Path to the executable file of the `Einit` initializing program.

- `--with-extra-asflags=FLAGS`

  Additional flags for the AS Assembler.

- `--with-extra-ldflags=FLAGS`

Additional flags for the LD Linker.

- `-h, --help`

  Displays the Help text.

Cross compilers

The toolchain provided in the KasperskyOS SDK includes one or more Clang compilers and a rustc compiler. The `toolchain/bin` directory contains the following files:

- Executable files of compilers (for example, `clang-17` or `rustc`)
- Executable files of linkers (for example, `aarch64-kos-ld`)
- Executable files of assemblers (for example, `aarch64-kos-as`)
- Bash scripts for the Clang compiler (for example, `aarch64-kos-clang`, `aarch64-kos-clang++`)

Linker operation specifics

When building the executable file of an application, by default the linker links the following libraries in the specified order:

1. libc is the standard C library.
2. libm is the library that implements the mathematical functions of the standard C language library.
3. libvfs_stubs is the library that contains stubs of I/O functions (for example, `open`, `socket`, `read`, `write`).
4. libkos is the library for accessing the KasperskyOS core endpoints.
5. libenv is the library of the subsystem for configuring the environment of applications (environment variables, arguments of the `main` function, and custom configurations).
6. libsrvtransport-u is the library that supports IPC between processes and the kernel.

Using dynamic libraries

Dynamic libraries (`*.so` files) can be used in a KasperskyOS-based solution. Compared to static libraries (`*.a` files), dynamic libraries provide the following advantages:

- Efficient use of RAM.

  Multiple processes can use the same instance of a dynamic library. Also, an executable file and dynamic libraries in one process can use the same instance of a dynamic library.

  Dynamic libraries can be loaded into memory and unloaded from memory on the initiative of the programs that use them.

- Convenient software updates.

A dynamic library update is applied to all executable files and dynamic libraries dependent on this dynamic library without having to rebuild them.

- Capability to implement a mechanism for plug-ins.

  Plug-ins for solution components consist of dynamic libraries.

- Shared use of code and data.

  One instance of a dynamic library can be concurrently used by multiple processes, and by an executable file and dynamic libraries in one process. This enables centralized management of multi-access to resources or storage of shared data, for example.

> Dynamic libraries are provided in KasperskyOS Community Edition, and can also be created by a KasperskyOS-based solution developer. Normal operation of third-party dynamic libraries cannot be guaranteed.
> Due to current technical limitations, `libc.so` and `libpthread.so` cannot be used in a KasperskyOS-based solution.

Prerequisites for using dynamic libraries

To use dynamic libraries in a KasperskyOS-based solution, the following conditions must be met:

1. Processes that use dynamic libraries must have access to the file systems in which the files of the dynamic libraries are stored. Access to file systems is provided by VFS, which is a separate process. The dynamic libraries must not be used by VFS or other software that is used by VFS to work with storage (such as a storage driver).

2. The system program `BlobContainer` must be included in the solution.

3. Executable files that use dynamic libraries must be built with the `-rdynamic` flag (with dynamic linking).

   The `CMake` `initialize_platform()` command causes this flag to be used automatically for building all executable files defined via `CMake` `add_executable()` commands.

> If the `CMake` command `initialize_platform(FORCE_STATIC)` is specified in the `CMakeLists.txt` root file, the toolchain performs static linking of executable files.
> The `CMake` `project_static_executable_header_default()` command affects the build of executable files defined via subsequent `CMake` `add_executable()` commands in one `CMakeLists.txt` file. The toolchain performs static linking of these executable files.
> The `CMake` `platform_target_force_static()` command affects the build of one executable file defined via the `CMake` `add_executable()` command. The toolchain performs static linking of this executable file.
> The executable file that is built with the `-rdynamic` flag is linked to a static library if a dynamic library is not found. For example, if the `CMake` `target_link_libraries` `(client -lm)` command is being used, the executable file of the `client` program is linked to the static library `libm.a` if the dynamic library `libm.so` is not found.

Life cycle of a dynamic library

The life cycle of a dynamic library includes the following phases:

1. Loading into memory.

   A dynamic library linked to an executable file is loaded into shared memory upon startup of the process that was created based on this executable file. A process can also load a dynamic library into shared memory by calling the `dlopen()` function of the POSIX interface. A dynamic library may be linked to other dynamic libraries, so an executable file depends not only on the dynamic library directly linked to it, but also depends on the entire dependency graph of this library. A dynamic library is loaded into shared memory together with all of the dynamic libraries that it depends on.

   One instance of a dynamic library is loaded into shared memory regardless of how many processes are using this library. (More specifically, only the part of the dynamic library that includes code and read-only data is loaded into shared memory. The other part of the dynamic library is loaded into the memory of each process that uses this library.) A dynamic library on which several other dynamic libraries depend is loaded into shared memory in a single instance.

   If a list of dynamic libraries is defined through the `LD_PRELOAD` environment variable, these dynamic libraries will be loaded into shared memory even if the executable file is not dependent on them. (List items must be absolute or relative paths to dynamic libraries separated by a colon, for example: `LD_PRELOAD=libmalloc.so:libfree.so:/usr/somepath/lib/libfoo.so`.) The functions that are exported by the dynamic libraries specified in `LD_PRELOAD` replace the identically named functions that are exported by other dynamic libraries. This can be used for debugging purposes if you need to replace functions imported from dynamic libraries.

   The dynamic library loader searches for dynamic libraries in the following order:

   1. Absolute paths defined through the `LD_LIBRARY_PATH` environment variable.

      Paths must be separated by a colon, for example: `LD_LIBRARY_PATH=/usr/lib:/home/user/lib`.

   2. Absolute paths defined in the `DT_RUNPATH` or `DT_RPATH` field of the `.dynamic` section of executable files and dynamic libraries.

      Linking of executable files and dynamic libraries may include defined paths that the dynamic library loader will search. (For example, this can be done through the `INSTALL_RPATH` property in the `CMake` command `set_target_properties()`.) Paths used to search for dynamic libraries are stored in the `DT_RUNPATH` or `DT_RPATH` field of the `.dynamic` section. This field may be in executable files linked to dynamic libraries and in dynamic libraries linked to other dynamic libraries.

   3. Path `/lib`.

   The dynamic library loader searches in this same order if a relative path to a dynamic library is specified in the `filename` parameter of the `dlopen()` function or in the `LD_PRELOAD` environment variable. If the absolute path is specified, the loader puts the dynamic library into shared memory without performing a search.

2. Use by a process (or processes).

3. Unloading from memory.

   A dynamic library is unloaded from shared memory when all processes using this library have terminated or called the `dlclose()` function of the POSIX interface. A dynamic library linked to an executable file cannot be unloaded from shared memory until termination of the process that was created based on this executable file. A dynamic library that is linked to other dynamic libraries is unloaded from shared memory after all libraries that depend on it are unloaded.

Including the BlobContainer system program in a KasperskyOS-based solution

If a solution uses dynamic libraries, the system program `BlobContainer` must be included in this solution (this system program is provided by the executable file `sysroot-aarch64-kos/bin/BlobContainer` from KasperskyOS Community Edition).

The `BlobContainer` program can be included in a solution either automatically or manually. This program is automatically included in a solution by running the `CMake` commands `build_kos_qemu_image()` and `build_kos_hw_image()` if at least one executable file in the solution is linked to a dynamic library. (To disable automatic inclusion of the `BlobContainer` program in a solution, you need to add the `NO_AUTO_BLOB_CONTAINER` value to the parameters of the `CMake` commands `build_kos_qemu_image()` and `build_kos_hw_image()`.) If programs in a solution work with dynamic libraries using only a POSIX interface (the `dlopen()`, `dlsym()`, `dlerror()`, and `dlclose()` functions), the `BlobContainer` program needs to be manually included in the solution.

The VFS working with files of dynamic libraries must be a separate process. You need to create an IPC channel from the process of the `BlobContainer` program to the VFS process.

You must create IPC channels from the processes using dynamic libraries to the process of the `BlobContainer` program. These IPC channels can be created statically or dynamically. If a statically created IPC channel is not available, the client and server parts of the `BlobContainer` program attempt to dynamically create an IPC channel using the name server.

If the `BlobContainer` program is automatically included in a solution, the `@INIT_EXTERNAL_ENTITIES@`, `@INIT_<program name>_ENTITY_CONNECTIONS@` and `@INIT_<program name>_ENTITY_CONNECTIONS+@` macros used in the `init.yaml.in` file automatically create within the init description dictionaries of IPC channels that enable static creation of IPC channels from processes created based on executable files linked to dynamic libraries to the process of the `BlobContainer` program. (The process of the `BlobContainer` program receives the name `kl.bc.BlobContainer`, while the IPC channels receive the name `kl.BlobContainer`.) However, dictionaries of IPC channels to the `BlobContainer` program process are not automatically created for processes that work with dynamic libraries using only a POSIX interface. To ensure that the required IPC channels are statically created, these dictionaries must be manually created (these IPC channels must have the name `kl.BlobContainer`).

If the `BlobContainer` program is manually included in the solution and you need to statically create IPC channels from processes using dynamic libraries to the `BlobContainer` program process, you must manually create dictionaries of the required IPC channels in the init description. By default, the IPC channel to the `BlobContainer` program process has the name `kl.BlobContainer`. However, this name can be changed through the environment variable `_BLOB_CONTAINER_BACKEND`. This variable must be defined for the `BlobContainer` process and for processes using dynamic libraries.

The environment variable `_BLOB_CONTAINER_BACKEND` defines not only the name of statically created IPC channels to the `BlobContainer` program process, but also defines the endpoint name that is published on the name server and used to dynamically create IPC channels to the `BlobContainer` program process. This is convenient when multiple processes of the `BlobContainer` program are running simultaneously (for example, to isolate its own dynamic libraries from external ones), and when different processes using dynamic libraries must interact over IPC with different processes of the `BlobContainer` program. In this case, you need to define different values for the environment variable `_BLOB_CONTAINER_BACKEND` for different processes of the `BlobContainer` program, and then use these values for the environment variable `_BLOB_CONTAINER_BACKEND` for processes using dynamic libraries. The specific value must be selected depending on the specific process of the `BlobContainer` program that requires the dynamically created IPC channel.

Example use of the environment variable `_BLOB_CONTAINER_BACKEND` in the `init.yaml.in` file:

```
entities:

- name: example.BlobContainer
  path: example_blob_container
  args:
  - "-v"
  env:
```

```
    _BLOB_CONTAINER_BACKEND: kl.custombc
@INIT_example_blob_container_ENTITY_CONNECTIONS@

- name: client.Client
  path: client
  env:
    _BLOB_CONTAINER_BACKEND: kl.custombc
@INIT_client_ENTITY_CONNECTIONS@

@INIT_EXTERNAL_ENTITIES@
```

Example use of the environment variable `_BLOB_CONTAINER_BACKEND` in `CMake` commands:

```
set_target_properties (ExecMgrEntity PROPERTIES
EXTRA_ENV
"    _BLOB_CONTAINER_BACKEND: kl.custombc")

set_target_properties (dump_collector::entity PROPERTIES
EXTRA_ENV
"    _BLOB_CONTAINER_BACKEND: kl.custombc")
```

Building dynamic libraries

To build a dynamic library, you need to use the following `CMake` command:

```
add_library(<build target name> SHARED <path to a file of the library source code>…)
```

You can also build a dynamic library by using the following `CMake` command:

```
add_library(<build target name> <path to a file of the library source code>…)
```

The `cmake` shell command must be called with the `-D BUILD_SHARED_LIBS=YES` parameter. (If the `cmake` shell command is called without the `-D BUILD_SHARED_LIBS=YES` parameter, a static library will be built.)

Example:

```
#!/bin/bash
…
cmake -G "Unix Makefiles" \
      -D CMAKE_BUILD_TYPE:STRING=Debug \
      -D CMAKE_TOOLCHAIN_FILE=$SDK_PREFIX/toolchain/share/toolchain-$TARGET.cmake \
      -D BUILD_SHARED_LIBS=YES \
      -B build \
      && cmake --build build --target kos-image
```

By default, the library file name matches the name of the build target defined via the parameter of the `CMake` `add_library()` command. The library file name can be changed by using the `CMake` `set_target_properties()` command. This can be done to make the library file name identical for its dynamic and static variants.

Example:

```
# Build the static library
add_library(somelib_static STATIC src/somesrc.cpp)
set_target_properties(somelib_static PROPERTIES OUTPUT_NAME "somelib")
# Build the dynamic library
```

```
add_library(somelib_shared SHARED src/somesrc.cpp)
set_target_properties(somelib_shared PROPERTIES OUTPUT_NAME "somelib")
```

A dynamic library can be linked to other static and dynamic libraries by using the `CMake` `target_link_libraries()` command. In this case, static libraries must be built with the `-fPIC` flag. This flag is applied when building a static library if the following `CMake` command is used:

```
set_property(TARGET <build target name>… PROPERTY POSITION_INDEPENDENT_CODE ON)
```

Adding dynamic libraries to a KasperskyOS-based solution

There are two ways to add dynamic libraries to a KasperskyOS-based solution: by either automatic or manual search for the dynamic libraries required by the solution programs.

Automatic search for dynamic libraries

Use the `PACK_DEPS_COPY_ONLY ON`, `PACK_DEPS_LIBS_PATH`, and `PACK_DEPS_COPY_TARGET` parameters in the `CMake` commands `build_kos_qemu_image()` and `build_kos_hw_image()`.

> Dynamic libraries that are loaded into memory by calling the `dlopen()` function of the POSIX interface will not be found by an automatic search.

Example 1:

```
# This example creates a solution image for the hardware platform.
# This solution image must include the storage drive image containing the dynamic
# libraries.

# Setting values of variables
set (IMAGE_FS ${CMAKE_BINARY_DIR}/hdd)
set (LIBS_PATH "${IMAGE_FS}/lib")
set (DISK_IMG ramdisk0.img)

if (blob_container_ENTITY_FOUND)
    # Configure the BlobContainer program
    set_target_properties (${blob_container_ENTITY} PROPERTIES
                           DEPENDS_ON_ENTITY ${precompiled_vfsVfsRamFs}
                           EXTRA_ENV "    VFS_FILESYSTEM_BACKEND: client:kl.VfsRamFs")
endif ()

# Create a storage drive image containing the dynamic libraries
# The storage drive image will be created after completing the copylibs target,
# which is created by the Cmake command build_kos_hw_image().
add_custom_command (OUTPUT ${DISK_IMG}
                    DEPENDS copylibs
                    COMMAND ${KL_SDK_ROOT_PATH}/common/prepare_hdd_img.sh
                            -d ${IMAGE_FS} -s 64 -f ext4 -img ${DISK_IMG}
                    COMMENT "Creating disk image '${DISK_IMG}' from files in '${IMAGE_FS}' ...")

# Create a solution image for the hardware platform
# The copylibs target indicates that the automatically found dynamic
# libraries will be copied to the directory ${LIBS_PATH}. This directory will be included
# in the storage drive image. The value ${DISK_IMG} is specified in the
# IMAGE_FILES parameter, so the storage drive image will be created before the solution image is created,
# and it will be included in the solution image.
build_kos_hw_image (kos-image
                    ...
                    IMAGE_FILES ${ENTITIES_LIST} ${DISK_IMG}
```

```
                    PACK_DEPS_COPY_ONLY ON
                    PACK_DEPS_LIBS_PATH ${LIBS_PATH}
                    PACK_DEPS_COPY_TARGET copylibs)
```

Example 2:

```
# This example to be run in QEMU creates a solution image and
# a separate storage drive image containing the dynamic libraries,
# and this example to be run on the hardware platform creates an SD card image
# to which the solution image and dynamic libraries are added.

# Set the values of variables used during the build
# for the hardware platform
set (HW_IMAGE_FS "${CMAKE_BINARY_DIR}/hdd_hw")
set (HW_LIBS_PATH "${HW_IMAGE_FS}/lib")

# Set the values of variables used during the build
# for QEMU
set (QEMU_IMAGE_FS "${CMAKE_BINARY_DIR}/hdd_qemu")
set (QEMU_LIBS_PATH "${QEMU_IMAGE_FS}/lib")
set (QEMU_DISK_IMAGE sdcard0.img)
set (QEMU_FLAGS "-nic none -m 2048 -drive file=${QEMU_DISK_IMAGE},if=sd,format=raw")
set (QEMU_DEPENDENCIES ${QEMU_DISK_IMAGE})

if (blob_container_ENTITY_FOUND)
    # Configure the BlobContainer program
    set_target_properties (${blob_container_ENTITY} PROPERTIES
                           DEPENDS_ON_ENTITY ${precompiled_vfsVfsSdCardFs}
                           EXTRA_ENV "    VFS_FILESYSTEM_BACKEND: client:kl.VfsSdCardFs")

# Create a storage drive image containing the dynamic libraries for QEMU
add_custom_command (${QEMU_DISK_IMAGE}
                    DEPENDS copylibs_qemu
                    COMMAND ${KL_SDK_ROOT_PATH}/common/prepare_hdd_img.sh
                            -d ${QEMU_IMAGE_FS} -s 64 -f fat32 -img ${QEMU_DISK_IMAGE}
                    COMMENT "Creating disk image '${QEMU_DISK_IMAGE}' from files in '${QEMU_IMAGE_FS}' ...")

# Create a solution image for QEMU
# The copylibs_qemu target indicates that the automatically found dynamic
# libraries will be copied to the directory ${QEMU_LIBS_PATH}. This directory will be included
# in the storage drive image containing the dynamic libraries for QEMU. The value ${QEMU_FLAGS}
# contains the -drive file=${QEMU_DISK_IMAGE},if=sd,format=raw parameters, which
# are required for QEMU use of the storage drive image containing the dynamic
# libraries. The value ${QEMU_DEPENDENCIES} is required to make sure that
# the storage drive image containing the dynamic libraries is created before creation of the
# solution image for QEMU. (The solution image will not include the storage drive image containing the
# dynamic libraries, but you must add this target dependency
# to create the storage drive image containing the dynamic libraries.)
build_kos_qemu_image (kos-qemu-image
                      ...
                      QEMU_FLAGS            "${QEMU_FLAGS}"
                      QEMU_DEPENDENCIES     "${QEMU_DEPENDENCIES}"
                      PACK_DEPS_COPY_ONLY   ON
                      PACK_DEPS_LIBS_PATH   "${QEMU_LIBS_PATH}"
                      PACK_DEPS_COPY_TARGET copylibs_qemu
                      IMAGE_FILES           ${ENTITIES})

# Create a solution image for the hardware platform
# The copylibs_hw target indicates that the automatically found dynamic
# libraries will be copied to the directory ${HW_LIBS_PATH}. This directory will be included
# in the SD card image that is created by the CMake command build_sd_image().
build_kos_hw_image (kos-image
                    ...
                    PACK_DEPS_COPY_ONLY   ON
                    PACK_DEPS_LIBS_PATH   "${HW_LIBS_PATH}"
                    PACK_DEPS_COPY_TARGET copylibs_hw
                    IMAGE_FILES           ${ENTITIES})

# The solution image for the hardware platform will be created after
# the automatically found dynamic libraries are copied to the directory ${HW_LIBS_PATH}.
# (The solution image will not include dynamic libraries, but you must add
# this target dependency to complete copying prior to creation of
# the SD card image.)
add_dependencies (kos-image copylibs_hw)

# Create the SD card image
# The SD card image will be created based on the directory ${HW_IMAGE_FS},
# which includes the solution for the hardware platform and the directory
# containing the dynamic libraries. The SD card image will be created
```

```
# after creation of the solution image for the hardware platform.
build_sd_image (sd-image
                KOS_IMAGE_TARGET kos-image
                IMAGE_FS         ${HW_IMAGE_FS})
```

Manual search for dynamic libraries

You need to independently search for dynamic libraries required by the solution programs. This way, even dynamic libraries that were loaded into memory by calling the `dlopen()` function of the POSIX interface can be added to a solution.

Example 1:

```
# This example creates a solution image that includes
# the libm.so dynamic library in addition to the executable files of programs.

# Configure the VFS program
set (VFS_SDCARD_ARGS "\
    - -l
    - nodev /tmp ramfs 0
    - -l
    - devfs /dev devfs 0
    - -l
    - romfs /lib romfs ro")
set_target_properties (${precompiled_vfsVfsSdCardFs} PROPERTIES
                       EXTRA_ARGS ${VFS_SDCARD_ARGS})

if (blob_container_ENTITY_FOUND)
    # Configure the BlobContainer program
    set_target_properties (${blob_container_ENTITY} PROPERTIES
                           DEPENDS_ON_ENTITY ${precompiled_vfsVfsSdCardFs}
                           EXTRA_ENV "    VFS_FILESYSTEM_BACKEND: client:kl.VfsSdCardFs")
endif ()

if(PLATFORM_SUPPORTS_DYNAMIC_LINKING)
    # Get the full path to the libm.so dynamic library
    find_file(LIBM_SO_FILE libm.so PATH_SUFFIXES lib REQUIRED)
    set(EXTRA_FILES ${LIBM_SO_FILE})
else()
    set(EXTRA_FILES)
endif()

# Create a solution image for the hardware platform
# The libm.so dynamic library will be added to the
# solution image because the value ${EXTRA_FILES}
# specified in the IMAGE_FILES parameter contains the full
# path to this library.
build_kos_hw_image(kos-image
                   ...
                   IMAGE_FILES ${ENTITIES}
                               ${EXTRA_FILES})
```

Example 2:

```
# This example to be run in QEMU creates a solution image and
# a separate storage drive image containing the libm.so dynamic library.

# Setting values of variables
set (QEMU_FLAGS "-nic none -m 2048")
set (QEMU_DEPENDENCIES)

if (PLATFORM_SUPPORTS_DYNAMIC_LINKING)
    # Setting values of variables
    set (IMAGE_FS "${CMAKE_BINARY_DIR}/hdd")
    set (LIBS_PATH "${IMAGE_FS}/lib")
    set (QEMU_DISK_IMAGE sdcard0.img)
    string (APPEND QEMU_FLAGS "-drive file=${QEMU_DISK_IMAGE},if=sd,format=raw")
    set (QEMU_DEPENDENCIES ${QEMU_DISK_IMAGE})

    # Configure the BlobContainer program
    set_target_properties (${blob_container_ENTITY} PROPERTIES
```

```
                            DEPENDS_ON_ENTITY ${precompiled_vfsVfsSdCardFs}
                            EXTRA_ENV "    VFS_FILESYSTEM_BACKEND: client:kl.VfsSdCardFs")

    # Get the full path to the libm.so dynamic library
    find_file (LIBM_SO_FILE libm.so PATH_SUFFIXES lib REQUIRED)

    # Create a storage drive image containing the libm.so dynamic library
    add_custom_command (OUTPUT ${QEMU_DISK_IMAGE}
                        COMMAND ${CMAKE_COMMAND} -E make_directory ${LIBS_PATH}
                        COMMAND ${CMAKE_COMMAND} -E copy ${LIBM_SO_FILE} ${LIBS_PATH}
                        COMMAND ${KL_SDK_ROOT_PATH}/common/prepare_hdd_img.sh
                                -d ${IMAGE_FS} -s 64 -f fat32 -img ${QEMU_DISK_IMAGE}
                        COMMENT "Creating disk image '${QEMU_DISK_IMAGE}' from files in '${IMAGE_FS}' ...")
endif ()

# Create a solution image for QEMU
# The value ${QEMU_FLAGS} contains the -drive file=${QEMU_DISK_IMAGE},if=sd,format=raw parameters,
# which are required for QEMU use of the storage drive image containing the
# libm.so dynamic library. The value ${QEMU_DEPENDENCIES} is required to make sure that
# the storage drive image containing the libm.so dynamic library is created before creation of the
# solution image for QEMU. (The solution image will not include the storage drive image containing the
# libm.so dynamic library, but you must add this target dependency
# to create the storage drive image containing the libm.so dynamic library.)
build_kos_qemu_image (kos-qemu-image
                      ...
                      QEMU_FLAGS        "${QEMU_FLAGS}"
                      QEMU_DEPENDENCIES "${QEMU_DEPENDENCIES}"
                      IMAGE_FILES       ${ENTITIES})
```

Debugging programs in a KasperskyOS-based solution

Using the GDB debugger

To debug programs in a KasperskyOS-based solution, you must use the GDB debugger from the KasperskyOS SDK on a computer running a Linux OS. Debugging can be performed in the QEMU from KasperskyOS Community Edition or on the hardware platform. To perform debugging in QEMU, you should use the GDB server of QEMU or the GDB server of the KasperskyOS kernel. To perform debugging on the hardware platform, you must use the GDB server of the KasperskyOS kernel.

> Correct operation of the GDB debugger and QEMU provided by KasperskyOS Community Edition cannot be guaranteed.
> By default, the GDB debugger supports OS ABI (Operating System ABI), which is specific to KasperskyOS. If the default OS ABI profile is modified, the capability for full-fledged debugging cannot be guaranteed.

You can debug multiple programs of a solution at the same time by loading debug symbols for multiple executable files.

The debug symbols of executable files can be saved in the executable files themselves or in separate files. In the latter case, you can significantly reduce the size of the solution image. To save debug symbols in `*.dbg` files, use the following `CMake` command:

```
add_custom_command(TARGET <program name> POST_BUILD
                   COMMAND ${CMAKE_OBJCOPY} --only-keep-debug $<TARGET_FILE:<program name>> $<TARGET_FILE_NAME:
<program name>>.dbg
                   COMMAND ${CMAKE_OBJCOPY} --strip-debug $<TARGET_FILE:<program name>>
                   COMMAND ${CMAKE_OBJCOPY} --add-gnu-debuglink=$<TARGET_FILE_NAME:<program name>>.dbg
$<TARGET_FILE:<program name>>)
```

Stack backtrace when a process abnormally terminates

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

> In some cases of abnormal process termination, the KasperskyOS kernel cannot prepare information about the call stack. For instance, this could be caused by optimization parameters that were applied when building a program that was run in the context of the unexpectedly terminated process.

Support for sanitizers

KasperskyOS Community Edition supports the ASAN and UBSAN sanitizers.

Debugging programs using the QEMU GDB server

The GDB server of QEMU has the following distinguishing characteristics:

- Does not recognize processes and threads managed by KasperskyOS.

  All processes managed by KasperskyOS are viewed as one process by the GDB server of QEMU. This process includes one or more threads depending on how many threads can be executed at the same time on the hardware platform emulated by QEMU. Therefore, the processes and threads handled by the GDB server of QEMU are not considered to be processes and threads managed by KasperskyOS.

- Does not guarantee debug capabilities for QEMU emulation of hardware platforms employing *symmetric multiprocessing* (SMP).

  The GDB server of QEMU can be used during QEMU emulation of hardware platforms employing SMP. However, GDB commands for step-by-step execution of a program may fail (for example, `next` and `step`).

Preparations for debugging

To prepare the conditions for debugging with the GDB server of QEMU, you must complete the following steps:

1. Disable ASLR support and define the memory addresses for loading the `.text` sections of executable files.

   To do so, you need to use the following `CMake` command:

   ```
   set_target_properties(<program name> PROPERTIES LINK_FLAGS "-no-pie -Ttext <address>")
   ```

   The GDB debugger needs the memory address used to load the ELF image to map the debug symbols to the ELF image. When ASLR is supported by default, this address is defined by a random value that is unknown to the GDB debugger, which complicates debugging. To disable ASLR support, indicate the `-no-pie` flag.

   The GDB server of QEMU views all running processes managed by KasperskyOS as one process. If ASLR is not supported, by default the linker causes the executable files to be loaded at the same memory address of processes. As a result, the loaded segments of executable files overlap in memory from the perspective of the GDB debugger, and debugging is carried out incorrectly (for example, incorrect values of variables are displayed, or breakpoints set for another program are triggered). To prevent overlapping of loaded segments corresponding to different programs, use the `-Ttext <address>` parameter that defines the load address of the `.text` section. The offset between the load addresses of the `.text` section in different processes can be a value that is 1 MB larger than the largest executable file in the solution image. For example, the value `0x06000000` can be specified as the loading address of the `.text` section for the `Client` program in the ping example, and the value `0x06200000` can be specified for the `Server` program in this example.

   > The offset between addresses of `.text` sections cannot guarantee that there will be no overlaps of loaded segments if pre-built executable files used in the solution (for example, from KasperskyOS Community Edition) were linked for ASLR support. (You can disable ASLR and define the addresses for loading `.text` sections only for the executable files that are built during creation of the solution. This cannot be done for pre-built executable files.) The `.text` sections of these executable files are loaded to random addresses, which could cause overlaps with loaded segments corresponding to the program that you need to debug.

2. [Optional] Define the settings for running QEMU.

   To change the initial settings and/or add new settings for running QEMU, you need to use the `QEMU_FLAGS` parameter of the `CMake` command `build_kos_qemu_image()`.

   Example:

   ```
   set(QEMU_FLAGS "-m 2048 -machine vexpress-a15,secure=on \
                   -cpu cortex-a72 -nographic -monitor none -smp 4")
   build_kos_qemu_image(kos-qemu-image
   ```

```
                    ...
                    QEMU_FLAGS "${QEMU_FLAGS}")
```

Initial steps of debugging

To begin debugging using the GDB server of QEMU, you must complete the following steps:

1. Build debug versions of executable files and libraries, create a solution image for QEMU, and run QEMU on standby for a connection from the GDB debugger.

   To do so, call the `cmake` shell commands, and specify the `-D CMAKE_BUILD_TYPE:STRING=Debug` and `--target gdbsim` parameters.

   Example:

   ```
   #!/bin/bash
   ...
   cmake -G "Unix Makefiles" \
         -D CMAKE_BUILD_TYPE:STRING=Debug \
         -D CMAKE_TOOLCHAIN_FILE=$SDK_PREFIX/toolchain/share/toolchain-$TARGET.cmake \
         -B build \
         && cmake --build build --target gdbsim
   ```

   Instead of the `gdbsim` target, you can specify the `gdbsim/fast` target to avoid rebuilding.

   QEMU starts but does not execute the solution code while it waits for a call of the GDB command `continue`.

2. Run the GDB debugger and connect to the GDB server of QEMU.

   To do so, you must call the shell command `make gdb` in the `build` directory.

3. [Optional] Define additional paths to search for dynamic libraries containing debug symbols.

   During step 1, the `build/einit/.gdbinit` file automatically records the path where the GDB debugger will search for dynamic libraries containing debug symbols from KasperskyOS Community Edition (as a parameter of the GDB command `set sysroot`). To define additional paths to search for dynamic libraries, call the following GDB command:

   ```
   set solib-search-path <path to directory>...
   ```

4. [Optional] Load debug symbols of executable files.

   If `.text` section loading addresses are defined for executable files with static linking (using the `CMake` command `set_target_properties()`), the GDB command for loading debug symbols for each of these files is automatically added to the `build/einit/.gdbinit` file at step 1.

   If a `.text` section loading address is defined for one executable file with dynamic linking, the GDB command for loading debug symbols for this file is automatically added to the `build/einit/.gdbinit` file at step 1. If `.text` section loading addresses are defined for multiple executable files with dynamic linking, the GDB command for loading debug symbols is not automatically added to the `build/einit/.gdbinit` file for any of these executable files at step 1.

   To manually load debug symbols, use the following GDB commands:

   - `add-symbol-file <path to file>` – for executable files with static or dynamic linking.
   - `file <path to file>` – for executable files with dynamic linking.

The GDB command `file` must be used so that the GDB debugger can load debug symbols of an executable file and the dynamic libraries required by this executable file. Use the GDB command `add-symbol-file` for the GDB debugger to load the debug symbols only for an executable file with dynamic linking.

The GDB command `file` can be used only for one executable file. In other words, the GDB debugger cannot load debug symbols of dynamic libraries for multiple executable files at the same time.

After the GDB command `file` is called, the following message can be displayed:

```
warning: Unable to find dynamic linker breakpoint function.GDB will be unable to debug shared library
initializers and track explicitly loaded dynamic code.
```

This message should be ignored.

If debug symbols are saved in separate files instead of in the executable files, links to the files containing the debug symbols are added to the executable files. When the GDB command `add-symbol-file` or `file` is called, you can specify the executable file or the file containing debug symbols.

> When performing a repeat build (step 1), you do not have to terminate the debug session (i.e. exit the debugger). (This means that you can avoid repeating operations such as loading debug symbols and defining additional paths to search for dynamic libraries.) To avoid terminating the debug session, you must run the GDB command `detach` before running the repeat build, then you must run the GDB command `target remote localhost:1234` after starting QEMU.
> GDB commands (for example, to load debug symbols or define additional paths to search for dynamic libraries) can be written to a file and then called by using the GDB command `source <path to file>`.

Special considerations for debugging emulated hardware platforms with SMP

Processors (computing kernels) are presented as threads.

To display information about processors, call the following GDB command:

```
info threads
```

To switch to another processor, call the following GDB command:

```
thread <Id value received by calling the GDB command info threads>
```

Debugging programs using the GDB server of the KasperskyOS kernel

The GDB server of the kernel has the following distinguishing characteristics:

- Recognizes processes and threads managed by KasperskyOS.
- Supports debugging on hardware platforms with SMP (including for emulation in QEMU).

Preparations for debugging in QEMU

To prepare the conditions for debugging with the GDB server of the KasperskyOS kernel in QEMU, you must complete the following steps:

1. Verify that KasperskyOS Community Edition contains a kernel version with the GDB server.

   To do so, make sure that the `libexec/aarch64-kos/kos-qemu-gdbstub` file is included in KasperskyOS Community Edition.

2. Add the `GDBSTUB_KERNEL` parameter to the list of parameters of the `CMake` command `build_kos_qemu_image()`.

   This parameter includes a kernel with the GDB server in the solution.

   Example:

```
build_kos_qemu_image(kos-qemu-image
                     GDBSTUB_KERNEL
                     ...
                     QEMU_FLAGS "${QEMU_FLAGS}")
```

3. Create a `.gdbinit` file so that you do not need to manually call the initial GDB commands every time the GDB debugger is started. (The `.gdbinit` file must be saved in the directory where the GDB debugger will be run.)

   Example:

```
# Define a path to search for dynamic libraries containing debug symbols from
# KasperskyOS Community Edition
set sysroot /opt/KasperskyOS-Community-Edition-<version>/sysroot-aarch64-kos
# Define additional paths to search for dynamic libraries containing debug
# symbols
set solib-search-path /home/user/example/build/lib1:/home/user/example/build/lib2
# Configure thread management (optional)
# The GDB debugger can operate in all-stop or non-stop mode.In the first mode,
# when the GDB debugger sees the stoppage of the thread whose context
# is in the debug focus, it assumes that all other threads
# known to the GDB debugger also stop. In the second mode, the GDB debugger assumes that
# all threads except the stopped thread continue to run. All-stop mode
# applied by the GDB debugger by default, and the GDB server of the kernel provides the capability to use
# only this mode. To edit the default settings
# for managing threads in all-stop mode, use GDB commands
# set scheduler-locking and set schedule-multiple.
# In this example, the first GDB command indicates that, when resuming
# execution of a thread whose context is in the debug focus (for example,
# the following GDB commands: continue, step, next), the GDB debugger must resume execution of all other
threads.
# The second GDB command indicates that the GDB debugger, when resuming execution of a thread
# whose context is in the debugging focus, must resume execution of all
# threads included in the processes of all programs that are being debugged.
set scheduler-locking off
set schedule-multiple on
# Connect the GDB debugger to the GDB server of the kernel
# The GDB server of the kernel interacts with the GDB debugger via an extended protocol,
# and QEMU provides access to the kernel GDB server via the TCP socket.
target extended-remote localhost:1234
```

Initial steps of debugging in QEMU

To begin debugging using the GDB server of the KasperskyOS kernel in QEMU, you must complete the following steps:

1. Build debug versions of executable files and libraries, create a solution image for QEMU, and run QEMU.

   To do so, call the `cmake` shell commands, and specify the `-D CMAKE_BUILD_TYPE:STRING=Debug` and `--target sim` parameters.

   Example:

   ```
   #!/bin/bash
   ...
   cmake -G "Unix Makefiles" \
         -D CMAKE_BUILD_TYPE:STRING=Debug \
         -D CMAKE_TOOLCHAIN_FILE=$SDK_PREFIX/toolchain/share/toolchain-$TARGET.cmake \
         -B build \
         && cmake --build build --target sim
   ```

   Instead of the `sim` target, you can specify the `sim/fast` target to avoid rebuilding.

   QEMU starts and executes the solution code. Execution of the solution code stops when the GDB debugger connects.

2. Run the GDB debugger and connect to the GDB server of the kernel.

   To do so, run the executable file `toolchain/bin/aarch64-kos-gdb` from KasperskyOS Community Edition. It must be run in the directory where the manually created `.gdbinit` file was saved.

3. Create inferiors, attach them to processes, and load debug symbols.

   To load debug symbols, you need to switch to the inferior associated with the process and use the following GDB commands:

   - `add-symbol-file <path to file>` — for executable files with static or dynamic linking.
   - `file <path to file>` – for executable files with dynamic linking.

   The GDB command `file` must be used so that the GDB debugger can load debug symbols of an executable file and the dynamic libraries required by this executable file. Use the GDB command `add-symbol-file` for the GDB debugger to load the debug symbols only for an executable file with dynamic linking.

   If debug symbols are saved in separate files instead of in the executable files, links to the files containing the debug symbols are added to the executable files. When the GDB command `add-symbol-file` or `file` is called, you can specify the executable file or the file containing debug symbols.

> When performing a repeat build (step 1), you need to terminate the debug session (exit the GDB debugger) and repeat steps 2 and 3.

Preparations for debugging on the hardware platform

To prepare the conditions for debugging with the GDB server of the KasperskyOS kernel on the hardware platform, you must complete the following steps:

1. Verify that KasperskyOS Community Edition contains kernel versions with the GDB server.

   To do so, make sure that the `libexec/aarch64-kos/kos-gdbstub` file is included in KasperskyOS Community Edition.

2. Establish a switched connection between the hardware platform and the computer where the GDB debugger will operate.

You must use USB-UART converters to connect the computer to the hardware platform. The diagrams for connecting USB-UART converters and hardware platforms are provided in the Preparing Raspberry Pi 4 B to run examples and Preparing Radxa ROCK 3A to run examples sections.

3. Add the `GDBSTUB_KERNEL` parameter to the list of parameters of the `CMake` command `build_kos_hw_image()`.

   This parameter includes a kernel with the GDB server in the solution.

   Example:

```
build_kos_hw_image(kos-image
                   GDBSTUB_KERNEL
                   ...
                   IMAGE_FILES ${ENTITIES})
```

4. Create a `.gdbinit` file so that you do not need to manually call the initial GDB commands every time the GDB debugger is started. (The `.gdbinit` file must be saved in the directory where the GDB debugger will be run.)

   Example:

```
# Define a path to search for dynamic libraries containing debug symbols from
# KasperskyOS Community Edition
set sysroot /opt/KasperskyOS-Community-Edition-<version>/sysroot-aarch64-kos
# Define additional paths to search for dynamic libraries containing debug
# symbols
set solib-search-path /home/user/example/build/lib1:/home/user/example/build/lib2
# Configure thread management (optional)
# The GDB debugger can operate in all-stop or non-stop mode.In the first mode,
# when the GDB debugger sees the stoppage of the thread whose context
# is in the debug focus, it assumes that all other threads
# known to the GDB debugger also stop. In the second mode, the GDB debugger assumes that
# all threads except the stopped thread continue to run. All-stop mode
# applied by the GDB debugger by default, and the GDB server of the kernel provides the capability to use
# only this mode. To edit the default settings
# for managing threads in all-stop mode, use GDB commands
# set scheduler-locking and set schedule-multiple.
# In this example, the first GDB command indicates that, when resuming
# execution of a thread whose context is in the debug focus (for example,
# the following GDB commands: continue, step, next), the GDB debugger must resume execution of all other
threads.
# The second GDB command indicates that the GDB debugger, when resuming execution of a thread
# whose context is in the debugging focus, must resume execution of all
# threads included in the processes of all programs that are being debugged.
set scheduler-locking off
set schedule-multiple on
# Define the symbol rate for interaction with the GDB server of the kernel
# You must set the value to 115200 baud.
set serial baud 115200
# Connect the GDB debugger to the GDB server of the kernel
# You must define the USB port that is used to connect the USB-UART converter
# used for debugging. You must also specify that the GDB server of the kernel
# interacts with the GDB debugger via an extended protocol.
target extended-remote /dev/ttyUSB1
```

Initial steps of debugging on the hardware platform

To begin debugging using the GDB server of the KasperskyOS kernel on the hardware platform, you must complete the following steps:

1. Build debug versions of executable files and libraries and create a solution image for the hardware platform.

   To do so, call the `cmake` shell commands and specify the `-D CMAKE_BUILD_TYPE:STRING=Debug` and `--target <tgt>` parameters, where `tgt` is the value of the `NAME` parameter in the `CMake` command `build_kos_hw_image()`.

Example:

```
#!/bin/bash
...
cmake -G "Unix Makefiles" \
      -D CMAKE_BUILD_TYPE:STRING=Debug \
      -D CMAKE_TOOLCHAIN_FILE=$SDK_PREFIX/toolchain/share/toolchain-$TARGET.cmake \
      -B build \
      && cmake --build build --target kos-image
```

2. Load the solution image to the hardware platform and start the solution.

   Execution of the solution code stops when the GDB debugger connects.

3. Run the GDB debugger and connect to the GDB server of the kernel.

   To do so, run the executable file `toolchain/bin/aarch64-kos-gdb` from KasperskyOS Community Edition. It must be run in the directory where the manually created `.gdbinit` file was saved.

4. Create inferiors, attach them to processes, and load debug symbols.

   To load debug symbols, you need to switch to the inferior associated with the process and use the following GDB commands:

   - `add-symbol-file <path to file>` — for executable files with static or dynamic linking.
   - `file <path to file>` – for executable files with dynamic linking.

   The GDB command `file` must be used so that the GDB debugger can load debug symbols of an executable file and the dynamic libraries required by this executable file. Use the GDB command `add-symbol-file` for the GDB debugger to load the debug symbols only for an executable file with dynamic linking.

   If debug symbols are saved in separate files instead of in the executable files, links to the files containing the debug symbols are added to the executable files. When the GDB command `add-symbol-file` or `file` is called, you can specify the executable file or the file containing debug symbols.

> When performing a repeat build (step 1), you need to terminate the debug session (exit the GDB debugger) and repeat steps 2–4.

Working with processes and threads

In the GDB debugger, objects called *inferiors* are used to work with processes. The process of each program to be debugged must be attached to a separate inferior. Switching between inferiors is equivalent to switching the debug focus to the context of another process. The debug symbols for the program to be debugged must be loaded after switching to the inferior that is attached to the process of this program.

An inferior is automatically created when the GDB debugger is started. When the GDB debugger connects to the GDB server of the kernel, execution of the solution code stops and this inferior is automatically attached to the process containing the thread in whose context the solution code execution is stopped. All other required inferiors can be created, attached to processes, and detached from processes manually or automatically. In the first case, you need to use the GDB commands for working with inferiors. In the second case, you must configure handling of events in the process life cycle.

GDB commands for working with inferiors

To work with inferiors, you need to use the following GDB commands:

- `info inferiors`

  Displays the list of inferiors. The current inferior is marked by the `*` character. (The context of the process associated with the current inferior is in the debug focus.)

- `add-inferior`

  Creates an inferior.

- `inferior <inferior number>`

  Switches to another inferior.

- `attach <PID>`

  Attaches the current inferior to the process with the specified PID. To print a list of processes, run the GDB command `info os processes`.

- `detach`

  Detaches the current inferior from the process.

Configuring handling of events in the process life cycle

The GDB server of the kernel notifies the GDB debugger about the occurrence of the following events:

1. Creation of a child process
2. Start of the initial thread of a child process
3. Termination of a process

The GDB debugger handles event 1 only if an inferior is attached to the parent process. To handle event 2, the GDB debugger must have previously handled event 1 when the child process was created. In this case, event 2 is handled even if the parent process was already detached from the inferior, for example, due to its termination.

To configure handling of events 1 and 2, use the following GDB commands:

- `set follow-fork-mode {child|parent}`

  Instructs the GDB debugger to switch to the context of the child process (`child`) or to remain in the context of the parent process (`parent`) when event 1 occurs. The second option is carried out by default.

- `set detach-on-fork {on|off}`

  Instructs the GDB debugger to leave the parent or child process attached to the inferior (`on`) or to leave the parent process attached to the inferior and create a new inferior and attach it to the child process (`off`) when event 1 occurs. The first option is carried out by default. In this case, the parameter of the GDB command `set follow-fork-mode` determines which process is attached to the inferior (parent or child).

- `set follow-exec-mode {new|same}`

  Instructs the GDB debugger to create a new inferior and attach it to the child process (`new`) or refrain from doing so (`same`) when event 2 occurs. The second option is carried out by default.

It is recommended to configure handling of events 1 and 2 as follows:

```
set follow-fork-mode parent
set follow-exec-mode same
set detach-on-fork off
```

This configuration ensures that events 1 and 2 are handled by the GDB debugger as follows:

- When event 1 occurs, a new inferior is created and attached to the child process.
- When event 1 occurs, the parent process remains attached to the inferior.
- When event 2 occurs, a new inferior is not created for the child process.

To avoid using an infinite loop, handling of event 2 by the GDB debugger can be used to stop execution of the program before management is transferred to the `main()` function.

To stop execution of the program when event 2 occurs, use the following GDB command:

```
catch exec
```

This GDB command stops the execution of each child process that was created while event 1 was handled. To stop the execution of individual processes, use the following GDB command:

```
monitor exec [<name of process 1>][, <name of process 2>]...
```

A subsequent call of this GDB command cancels the previous call, and calling this command without parameters will stop each child process that was created while event 1 was handled.

Event 3 is handled by the GDB debugger only if inferiors attached to terminated processes are created. When this event is handled, the GDB debugger detaches the inferior from the terminated process. Handling of event 3 does not need to be configured.

GDB commands for working with threads

To print a list of the threads contained in processes attached to inferiors, run the following GDB command:

```
info threads
```

The current thread is marked by the `*` character. (The context of the current thread is in the debug focus.)

To switch to the context of another thread, use the following GDB command:

```
thread <Id value received by calling the GDB command info threads>
```

Switching to the context of another thread will result in switching to another inferior if this thread is in a process attached to an inferior different from the current one.

Analyzing stack backtrace data when a process abnormally terminates

When a process is terminated due to an unhandled exception, the KasperskyOS kernel prints stack backtrace data in the following format:

```
Call Trace (User Mode):
    [<0000000004068498>] (__assert_func+0x78) from [<000000000405f478>] (_vfs_backend_create+0xe8)
    [<000000000405f478>] (_vfs_backend_create+0xe8) from [<00000000040bdab0>] (_vfs_init+0xd0)
    [<00000000040bdab0>] (_vfs_init+0xd0) from [<00000000040111a8>] (init_vfs_helper+0xc8)
    [<00000000040111a8>] (init_vfs_helper+0xc8) from [<00000000040117e0>] (__eprol+0x88)
    [<00000000040117e0>] (__eprol+0x88)
```

Here you can clearly see the sequence of function calls that led to the unexpected termination of the process. At the end of the sequence, the `_vfs_backend_create()` function code residing in the process memory at the address `0x000000000405f478` called the `__assert_func()` function, and the `__assert_func()` function code residing in the process memory at the address `0x0000000004068498` invoked the exception. In other words, the exception occurred during verification of the conditions in the `_vfs_backend_create()` function. (An entry in the format `<function name>+0x<offset>` indicates an address within the function code. For example, the entry `_vfs_backend_create+0xe8` indicates an address within the `_vfs_backend_create()` function code offset by `0xe8` bytes relative to the address of the start of this function's code. The entry `_vfs_backend_create+0xe8` corresponds to the address `0x000000000405f478`.)

The names of all or some functions in the sequence of calls may fail to appear, which makes it more difficult to determine the cause of the abnormal process termination.

Example stack backtrace data containing only memory addresses:

```
Call Trace (User Mode):
    [<0000000004068498>] from [<000000000405f478>]
    [<000000000405f478>] from [<00000000040bdab0>]
    [<00000000040bdab0>] from [<00000000040111a8>]
    [<00000000040111a8>] from [<00000000040117e0>]
    [<00000000040117e0>]
```

The reason for the missing names of functions in the stack backtrace data is as follows. To print stack backtrace data, the kernel gets the names of functions from the symbol table `.symtab` and the string table `.strtab` that are loaded into the memory of the abnormally terminated process from the debug version of the executable file. These tables may be incomplete or entirely missing from the memory of the abnormally terminated process. In addition, the kernel cannot print the names of functions that were imported from dynamic libraries.

The symbol table `.symtab` and string table `.strtab` are absent from process memory in the following cases:

- The process was created from the normal release version of the executable file. (The executable file was built as the normal release version, and the normal release versions of static libraries were included in it during the build.)

  In contrast to the debug version, the normal release version of the executable file does not contain the symbol table `.symtab` and string table `.strtab`.

- The process was created from the debug version of an executable file from which the symbol table `.symtab` and string table `.strtab` were deleted before the file was added to the solution image. For instance, they could have been deleted by the `strip` utility.

- The process was created from the debug version of an executable file from which the symbol table `.symtab` and string table `.strtab` were not deleted before the file was added to the solution image, but these tables were not loaded into the process memory.

  The high-level API for process management and the low-level API for process management let you create a process with optional loading of the symbol table `.symtab` and string table `.strtab`.

The symbol table `.symtab` and string table `.strtab` are incomplete in the following cases:

- If the executable file was built as a debug version, the normal release versions of static libraries were included in the file during the build, and the symbol table `.symtab` and string table `.strtab` will not contain information about the functions implemented in these libraries.

- If the executable file was built as a normal release version, the debug versions of static libraries were included in the file during the build, and the symbol table `.symtab` and string table `.strtab` will contain information only about the functions that are implemented in these libraries.

For some memory addresses specified in stack backtrace data, information about functions can be obtained from the executable file. If a memory address is present in the symbol table `.symtab` of the executable file, you can get the code line number and the name of the function corresponding to this memory address. If a memory address is missing from the symbol table `.symtab` but is present in the symbol table `.dynsym` of the executable file, you can get the name of the function corresponding to this memory address. (When using ASLR, the memory addresses specified in the stack backtrace data are increased by the `Relocation base` value for the ELF image load offset that is printed together with the stack backtrace data.) Otherwise, information about functions cannot be received from the executable file. For instance, this applies to functions imported from dynamic libraries and to functions implemented in normal release versions of static libraries.

An executable file in KasperskyOS contains a set of exportable functions (but is not a dynamic library). For example, the functions of the `libc` library included in executable files are exportable. The symbol table `.dynsym` contains information about exportable functions. This table is present in the release version of the executable file, and remains in the debug version of the executable file after the debug symbols are deleted from it. (The symbol table `.symtab` and string table `.strtab` are part of the debug symbols.)

To get information about functions corresponding to the memory addresses specified in backtrace data from an executable file, use the `addr2line` utility provided in KasperskyOS Community Edition (the `toolchain/bin/aarch64-kos-addr2line` executable file). Format of the command for starting the `addr2line` utility:

```
aarch64-kos-addr2line [--functions] --exe=<path to the executable file> 0x<address 1> [0x<address 2>]...
```

In this command, you need to specify the memory addresses from the stack backtrace data reduced by the `Relocation base` value. The `--functions` parameter requires output of the names of functions in addition to the numbers of source code lines.

Example:

```
$ aarch64-kos-addr2line --functions --exe=./build/hello/Hello 0x30090
main
/opt/KasperskyOS-Community-Edition-<version>/examples/hello/hello/src/hello.c:6
```

Formal specifications of KasperskyOS-based solution components

Solution development includes the creation of formal specifications for its components that form a global picture for the Kaspersky Security Module. A *formal specification of a KasperskyOS-based solution component* (hereinafter referred to as the *formal specification of the solution component*) is comprised of a system of IDL, CDL and EDL descriptions (IDL and CDL descriptions are optional) for this component. These descriptions are used to automatically generate transport code of solution components, and source code of the security module and the initializing program. The formal specifications of solution components are also used as source data for the solution security policy description.

> Just like solution components, the KasperskyOS kernel also has a formal specification (for details, see Methods of KasperskyOS core endpoints).

Each solution component corresponds to an EDL description. In terms of a formal specification, a solution component is a container for components that provide endpoints. Multiple instances of one solution component may be used at the same time, which means that multiple processes can be started from the same executable file. Processes that correspond to the same EDL description are processes of the same class. An EDL description defines the process class name and the top-level component parameters, such as the provided endpoints with one or multiple interfaces, the security interface, and embedded components.

Each embedded component corresponds to a CDL description. This description defines the component name, provided endpoints, security interface, and embedded components. Embedded components can simultaneously provide endpoints, support a security interface, and serve as containers for other components. Each embedded component can provide multiple endpoints with one or more interfaces.

Each interface (including the security interface) is defined in an IDL description. This description defines the interface name, signatures of interface methods, and data types for the parameters of interface methods. The data comprising signatures of interface methods and definitions of data types for parameters of interface methods is referred to as a package.

Processes that do not provide endpoints may only act as clients. Processes that provide endpoints are servers, but they can also act as clients at the same time.

> The formal specification of a solution component does not define how this component will be implemented. In other words, the presence of components in a formal specification of a solution component does not mean that these components will be present in the architecture of this solution component.

Names of process classes, components, packages and interfaces

Process classes, components, packages and interfaces are identified by their names in IDL, CDL and EDL descriptions. Within one KasperskyOS-based solution, the names of process classes and the names of components form one set of names, while the names of packages form a different set of names. These two sets may overlap. A set of package names includes a set of interface names.

The name of a process class, component, package or interface is a link to the IDL, CDL or EDL file in which this name is defined. This link is a path to the IDL, CDL or EDL file (without the extension and dot before it) relative to the directory that is included in the set of directories where the source code generators search for IDL, CDL and EDL files. (This set of directories is defined by parameters `-I <path to the directory>`.) A dot is used as a separator in a path description.

For example, the `kl.core.NameServer` process class name is a link to the EDL file named `NameServer.edl`, which is located in the KasperskyOS SDK at the following path:

`sysroot-*-kos/include/kl/core`

However, source code generators must be configured to search for IDL, CDL and EDL files in the following directory:

`sysroot-*-kos/include`

> The name of an IDL, CDL or EDL file begins with an uppercase letter and must not contain any underscores `_`.

EDL description

EDL descriptions are placed into separate `*.edl` files and contain declarations in the Entity Definition Language (EDL):

1. **Process class name.** Uses any of the following declarations:

```
task class <process class name>
// Obsolete and not recommended for use.
entity <process class name>
```

2. [Optional] **List of instances of components.** The following declaration is used:

```
components {
    <component instance name> : <component name>
    ...
}
```

   Each component instance is indicated in a separate line. The component instance name must not contain any underscores `_`. The list can contain multiple instances of one component. Each component instance in the list has a unique name.

3. [Optional] **Security interface.** The following declaration is used:

```
security <interface name>
```

4. [Optional] **List of endpoints.** The following declaration is used:

```
endpoints {
    <endpoint name : interface name>
    ...
}
```

Each endpoint is indicated in a separate line. The endpoint name must not contain any underscores `_`. The list can contain multiple endpoints with the same interface. Each endpoint in the list has a unique name.

The EDL language is case sensitive.

Single-line comments and multi-line comments can be used in an EDL description.

> A security interface and provided endpoints can be defined in an EDL description and in a CDL description. If solution component development is utilizing already prepared constituent parts (such as libraries) that are accompanied by CDL descriptions, it is advisable to refer to them from the EDL description by using the `components` declaration. Otherwise, you can describe all provided endpoints in the EDL description. In addition, you can separately define the security interface in the EDL description and in each CDL description.

Examples of EDL files

Hello.edl

```
// Class of processes that do not contain components.
task class Hello
```

Signald.edl

```
/* Class of processes that contain
 * one instance of one component. */
task class kl.Signald

components {
    signals : kl.Signals
}
```

LIGHTCRAFT.edl

```
/* Class of processes that contain
 * two instances of different components. */
task class kl.drivers.LIGHTCRAFT

components {
    KUSB : kl.drivers.KUSB
    KIDF : kl.drivers.KIDF
}
```

Downloader.edl

```
/* Class of processes that do not contain
 * components and provide one endpoint. */
task class updater.Downloader

endpoints {
    download : updater.Download
}
```

CDL description

CDL descriptions are placed into individual `*.cdl` files and contain declarations in the Component Definition Language (CDL):

1. **The name of the component.** The following declaration is used:

```
component <component name>
```

2. [Optional] **Security interface.** The following declaration is used:

```
security <interface name>
```

3. [Optional] **List of endpoints.** The following declaration is used:

```
endpoints {
    <endpoint name : interface name>
    ...
}
```

   Each endpoint is indicated in a separate line. The endpoint name must not contain any underscores `_`. The list can contain multiple endpoints with the same interface. Each endpoint in the list has a unique name.

4. [Optional] **List of instances of embedded components.** The following declaration is used:

```
components {
    <component instance name : component name>
    ...
}
```

   Each component instance is indicated in a separate line. The component instance name must not contain any underscores `_`. The list can contain multiple instances of one component. Each component instance in the list has a unique name.

The CDL language is case sensitive.

Single-line comments and multi-line comments can be used in a CDL description.

> At least one optional declaration is used in a CDL description. If a CDL description does not use at least one optional declaration, this description will correspond to an "empty" component that does not provide endpoints, does not contain embedded components, and does not support a security interface.

Examples of CDL files

KscProductEventsProvider.cdl

```
// Component provides one endpoint.
component kl.KscProductEventsProvider

endpoints {
    eventProvider : kl.IKscProductEventsProvider
}
```

KscConnectorComponent.cdl

```
// Component provides multiple endpoints.
component kl.KscConnectorComponent

endpoints {
    KscConnCommandSender : kl.IKscConnCommandSender
    KscConnController : kl.IKscConnController
    KscConnSettingsHolder : kl.IKscConnSettingsHolder
    KscDataProvider : kl.IKscDataProvider
    ProductDataHolder : kl.IProductDataHolder
    KscDataNotifier : kl.IKscDataNotifier
    KscConnectorStateNotifier : kl.IKscConnectorStateNotifier
}
```

FsVerifier.cdl

```
/* Component does not provide endpoints, supports
 * a security interface, and contains one instance
 * of another component. */
component FsVerifier

security Approve

components {
    verifyComp : Verify
}
```

IDL description

IDL descriptions are placed into separate `*.idl` files and contain declarations in the Interface Definition Language (IDL):

1. **Package name.** The following declaration is used:

   ```
   package <package name>
   ```

2. [Optional] **Packages from which the data types for interface method parameters are imported.** The following declaration is used:

   ```
   import <package name>
   ```

3. [Optional] **Definitions of data types for parameters of interface methods**.

4. [Optional] **Signatures of interface methods.** The following declaration is used:

   ```
   interface {
       <interface method name>([<parameters>]);
       ...
   }
   ```

   Each method signature is indicated in a separate line. The method name must not contain any underscores `_`. Each method in the list has a unique name. The parameters of methods are divided into input parameters (`in`), output parameters (`out`), and parameters for transmitting error information (`error`). The order of parameters in the description is important: first input parameters, then output parameters, then error parameters. Methods of the security interface cannot have output parameters and error parameters.

   Input parameters and output parameters are transmitted in IPC requests and IPC responses, respectively. Error parameters are transmitted in IPC responses if the server cannot correctly handle the corresponding IPC requests.

The server can inform a client about IPC request processing errors via error parameters as well as through output parameters of interface methods. If the server sets the error flag in an IPC response when an error occurs, this IPC response will contain the error parameters without any output parameters. Otherwise this IPC response will contain output parameters just like when requests are correctly processed. (The error flag is set in IPC responses by using the `nk_err_reset()` macro defined in the `nk/types.h` header file from the KasperskyOS SDK.)

An IPC response sent with the error flag set and an IPC response with the error flag not set are considered to be different types of events for the Kaspersky Security Module. When describing a solution security policy, this difference lets you conveniently distinguish between the processing of events associated with the correct execution of IPC requests and the processing of events associated with incorrect execution of IPC requests. If the server does not set the error flag in IPC responses, the security module must check the values of output parameters indicating errors to properly process events related to the incorrect execution of IPC requests. (A client can check the state of the error flag in an IPC response even if the corresponding interface method does not contain error parameters. To do so, the client uses the `nk_msg_check_err()` macro defined in the `nk/types.h` header file from the KasperskyOS SDK.)

Signatures of interface methods cannot be imported from other IDL files.

The IDL language is case sensitive.

Single-line comments and multi-line comments can be used in an IDL description.

> At least one optional declaration is used in a IDL description. If an IDL description does not use at least one optional declaration, this description will correspond to an "empty" package that does not assign any interface methods or data types (including from other IDL descriptions).
> Some IDL files from the KasperskyOS SDK do not describe interface methods, but instead only contain definitions of data types. These IDL files are used only as exporters of data types.
> If a package contains a description of interface methods, the interface name matches the package name.

Examples of IDL files

Env.idl

```
package kl.Env

// Definitions of data types for interface method parameters
typedef string<128> Name;
typedef string<256> Arg;
typedef sequence<Arg,256> Args;

// Interface includes one method.
interface {
    Read(in Name name, out Args args, out Args envs);
}
```

Kpm.idl

```
package kl.Kpm

// Import data types for parameters of interface methods
import kl.core.Types

// Definition of data type for parameters of interface methods
typedef string<64> String;

/* Interface includes multiple methods.
 * Some methods do not have any parameters. */
interface {
    Shutdown();
    Reboot();
    PowerButtonPressedWait();
    TerminationSignalWait(in UInt32 entityId, in String entityName);
```

```
    EntityTerminated(in UInt32 entityId);
    Terminate(in UInt32 callingEntityId);
}
```

MessageBusSubs.idl

```
package kl.MessageBusSubs

// Import data types for interface method parameters
import kl.MessageBusTypes

/* Interface includes a method that has
 * input and output parameters, and
 * an error parameter.*/
interface {
    Wait(in ClientId id,
         out Message topic,
         out BundleId dataId,
         error ResultCode result);
}
```

WaylandTypes.idl

```
// Package contains only definitions of data types.
package kl.WaylandTypes

typedef UInt32                    ClientId;
typedef bytes<8192>               Buffer;
typedef string<4096>              ConnectionId;
typedef SInt32                    SsizeT;
typedef UInt32                    SizeT;
typedef SInt32                    ShmFd;
typedef SInt32                    ShmId;
typedef bytes<16384000>           ShmBuffer;
```

IDL data types

IDL supports primitive data types as well as composite data types. The set of supported composite types includes unions, structures, arrays, and sequences.

Primitive types

IDL supports the following primitive types:

- `SInt8`, `SInt16`, `SInt32`, `SInt64` – signed integer.
- `UInt8`, `UInt16`, `UInt32`, `UInt64` – unsigned integer.
- `Handle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field.
- `bytes<<size in bytes>>` – byte buffer consisting of a memory area with a size that does not exceed the defined number of bytes.
- `string<<size in bytes>>` – string buffer consisting of a byte buffer whose last byte is a terminating zero. The maximum size of a string buffer is a unit larger than the defined size due to the additional byte with the terminating zero.

Integer literals can be specified in decimal format, hexadecimal format (for example, `0x2f`, `0X2f`, `0x2F`, `0X2F`) or octal format (for example, `00123`, `0o123`).

You can use the reserved word `const` to define the named integer constants by assigning their values using integer literals or integer expressions.

Example definitions of named integer constants:

```
const UInt32 DeviceNameMax          = 0o100;
const UInt32 HandleTypeUserLast     = 0x0001FFFF;
const UInt32 MaxLogMessageSize = (2 << 3) ** 2;
const UInt32 MaxLogMessageCount = 100;
const UInt64 MaxLen = (MaxLogMessageSize + 4) * MaxLogMessageCount;
```

Named integer constants can be used to avoid problems associated with so-called "magic numbers". For example, if an IDL description defines named integer constants for return codes of an interface method, you can interpret these codes without additional information when describing a policy. Named integer constants and integer expressions can also be applied in definitions of byte buffers, string buffers, and composite types to define the size of data or the number of data elements.

The `bytes<<size in bytes>>` and `string<<size in bytes>>` constructs are used in definitions of composite types, signatures of interface methods, and when creating type aliases because they define anonymous types (types without a name).

Unions

A union stores different types of data in one memory area. In an IPC message, a union is provided with an additional `tag` field that defines which specific member of the union is used.

The following construct is used to define a union:

```
union <type name> {
    <member type> <member name>;
    ...
}
```

Example of a union definition:

```
union ExitInfo {
    UInt32        code;
    ExceptionInfo exc;
}
```

Structures

The following construct is used to define a structure:

```
struct <type name> {
    <field type> <field name>;
    ...
}
```

Example of a structure definition:

```
struct SessionEvqParams {
    UInt32 count;
    UInt32 align;
    UInt32 size;
}
```

Arrays

The following construct is used to define an array:

```
array<<type of elements>, <number of elements>>
```

This construct is used in definitions of other composite types, signatures of interface methods, and when creating type aliases because it defines an anonymous type.

> The `Handle` type can be used as the type of array elements if this array is not included in another composite data type. However, the total number of handles in an IPC message cannot exceed 255.

Sequences

A sequence is a variable-sized array. When defining a sequence, the maximum number of elements of the sequence is specified.

The following construct is used to define a sequence:

```
sequence<<type of elements>, <number of elements>>
```

This construct is used in definitions of other composite types, signatures of interface methods, and when creating type aliases because it defines an anonymous type.

> The `Handle` type cannot be used as the type of sequence elements.

Variable-size and fixed-size types

The `bytes`, `string` and `sequence` types are variable-size types. In other words, the maximum number of elements is assigned when defining these types, but less elements (or none) may actually be used. Data of the `bytes`, `string` and `sequence` types are stored in the IPC message arena. All other types are fixed-size types. Data of fixed-size types are stored in the constant part of IPC messages.

Types based on composite types

Composite types can be used to define other composite types. The definition of an array or sequence can also be included in the definition of another type.

Example definition of a structure with embedded definitions of an array and sequence:

```
const UInt32 MessageSize = 64;

struct BazInfo {
    array<UInt8, 100> a;
    sequence<sequence<UInt32, MessageSize>, ((2 << 2) + 2 ** 2) * MessageSize> b;
    string<100> c;
    bytes<4096> d;
    UInt64 e;
}
```

The definition of a union or structure cannot be included in the definition of another type. However, a type definition may include already defined unions and structures. This is done by indicating the names of the included types in the type definition.

Example definition of a structure that includes a union and structure:

```
union foo {
    UInt32 value1;
    UInt8 value2;
}

struct bar {
    UInt32 a;
    UInt8 b;
}

struct BazInfo {
    foo x;
    bar y;
}
```

Creating aliases of types

Type aliases make it more convenient to work with types. For example, type aliases can be used to assign mnemonic names to types that have abstract names. Assigned aliases for anonymous types also let you receive named types.

The following construct is used to create a type alias:

```
typedef <type name/anonymous type definition> <type alias>
```

Example of creating mnemonic aliases:

```
typedef UInt64 ApplicationId;
typedef Handle PortHandle;
```

Example of creating an alias for an array definition:

```
typedef array<UInt8, 4> IP4;
```

Example of creating an alias for a sequence definition:

```
const UInt32 MaxDevices = 8;
struct Device {
    string<32> DeviceName;
    UInt8 DeviceID;
}
```

```
typedef sequence<Device, MaxDevices> Devices;
```

Example of creating an alias for a union definition:

```
union foo {
    UInt32 value1;
    UInt8 value2;
}

typedef foo bar;
```

Defining anonymous types in signatures of interface methods

Anonymous types can be defined in signatures of interface methods.

Example of defining a sequence in an interface method signature:

```
const UInt8 DeviceCount = 8;

interface {
    Poll(in UInt32 timeout,
         out sequence<UInt32, DeviceCount / 2> report,
         out UInt32 count,
         out UInt32 rc);
}
```

Integer expressions in IDL

Integer expressions in IDL are composed of named integer constants, integer literals, operators (see the table below), and grouping parentheses.

Example use of integer expressions:

```
const UInt8 itemHeaderLen = 2;
const UInt8 itemBlockLen = 4;
const UInt8 maxItemCount = 0X10;
const UInt64 maxLen = (2 << 3) + (itemHeaderLen + itemBlockLen * 4) * maxItemCount;

interface {
    CopyPage(in array<UInt8, 4 * maxLen> page);
}
```

If an integer overflow occurs when computing an expression, the source code generator using the IDL file will terminate with an error.

Details on operators of integer expressions in IDL

Syntax	Operation	Precedence	Associativity	Special considerations
-a	Sign change	1	No	N/A
~a	Bitwise negation	1	No	N/A
a ** b	Exponentiation	2	No	Special considerations: • Due to the lack of associativity, you must use parentheses when specifying multiple consecutive operators to define the order of operations.

				Example:
				`(a ** b) ** c`
				`a ** (b ** c)`
				• The power exponent cannot be negative.

`a * b`	Multiplication	3	Left	N/A
`a / b`	Integer division	3	Left	Special considerations: • The result of the operation is a value obtained by rounding the real quotient down to the next lower integer. For example, *4 / 3 = 1, -4 / 3 = -2*. • The divisor cannot be zero.
`a % b`	Modulo	3	Left	Special considerations: • The sign of the operation result matches the sign of the divisor. For example, *-5 % 2 = 1, 5 % -2 = -1*. • The divisor cannot be zero.
`a + b`	Addition	4	Left	N/A
`a - b`	Subtraction	4	Left	N/A

`a << b`	Bit shift left	2*	No	Special considerations:
				• The precedence is lower than with unary operations but incomparable to the precedences of binary arithmetic operations. Therefore, in expressions with binary arithmetic operations, you must use parentheses to define the order of operations.
				Example:
				`a << (b + c)`
				`(a << b) ** c`
				• Due to the lack of associativity, you must use parentheses when specifying multiple consecutive operators to define the order of operations.
				Example:
				`(a << b) << c`
				`a << (b << c)`
				• The shift value must be within the interval *[0; 63]*.

`a >> b`	Bit shift right	2*	No	Special considerations:
				• The precedence is lower than with unary operations but incomparable to the precedences of binary arithmetic operations. Therefore, in expressions with binary arithmetic operations, you must use parentheses to define the order of operations.
				Example:
				`a >> (b * c)`
				`(a >> b) / c`
				• Due to the lack of associativity, you must use parentheses when specifying multiple consecutive operators to define the order of operations.
				Example:
				`(a >> b) >> c`
				`a >> (b >> c)`
				• The shift value must be within the interval *[0; 63]*.

Describing a security policy for a KasperskyOS-based solution

A *KasperskyOS-based solution security policy description* (hereinafter also referred to as a *solution security policy description* or *policy description*) provides a set of interrelated text files with the `psl` extension that contain declarations in the PSL language (Policy Specification Language). Some files reference other files through an inclusion declaration, which results in a hierarchy of files with one top-level file. The top-level file is specific to the solution. Files of lower and intermediate levels contain parts of the solution security policy description that may be specific to the solution or may be re-used in other solutions.

Some of the files of the lower and intermediate levels are provided in the KasperskyOS SDK. These files contain definitions of the basic data types and formal descriptions of KasperskyOS security models. *KasperskyOS security models* (hereinafter referred to as *security models*) serve as the framework for implementing security policies for KasperskyOS-based solutions. Files containing formal descriptions of security models reference a file containing definitions of the basic data types that are used in the descriptions of models.

The other files of lower and intermediate levels are created by the policy description developer if any parts of the policy description need to be re-used in other solutions. A policy description developer can also put parts of the policy description into separate files for convenient editing.

The top-level file references files containing definitions of basic data types and descriptions of security models that are applied in the part of the solution security policy that is described in this file. The top-level file also references all files of the lower and intermediate levels that were created by the policy description developer.

> The top-level file is normally named `security.psl`, but it can have any other name in the `*.psl` format.

General information about a KasperskyOS-based solution security policy description

In simplified terms, a KasperskyOS-based solution security policy description consists of bindings that associate descriptions of security events with calls of methods provided by security model objects. A *security model object* is an instance of a class whose definition is a formal description of a security model (in a PSL file). Formal descriptions of security models contain signatures of *methods of security models* that determine the permissibility of interactions between different processes and between processes and the KasperskyOS kernel. These methods are divided into two types:

- *Security model rules* are methods of security models that return a "granted" or "denied" result. Security model rules can change security contexts (for information about a security context, see Resource Access Control).
- *Security model expressions* are methods of security models that return values that can be used as input data for other methods of security models.

A security model object provides methods that are specific to one security model and stores the parameters used by these methods (for example, the initial state of a finite-state machine or the size of a container for specific data). The same object can be used to work with multiple resources. (In other words, you do not need to create a separate object for each resource.) However, the security contexts of these resources will be independent of each other. Likewise, multiple objects of one or more different security models can be used to work with the same resource. In this case, different objects will use the security context of the same resource without any reciprocal influence.

Security events serve as signals indicating the initiation of interaction between different processes and between processes and the KasperskyOS kernel. Security events include the following events:

- Clients send IPC requests.
- Servers or the kernel send IPC responses.
- The kernel or processes initialize the startup of processes.
- The kernel starts.
- Processes query the Kaspersky Security Module via the security interface.

Security events are processed by the security module.

Security models

The KasperskyOS SDK provides PSL files that describe the following security models:

- Base – methods that implement basic logic.
- Pred – methods that implement comparison operations.
- Bool – methods that implement logical operations.
- Math – methods that implement integer arithmetic operations.
- Struct – methods that provide access to structural data elements (for example, access to parameters of interface methods transmitted in IPC messages).
- Regex – methods for text data validation based on regular expressions.
- HashSet – methods for working with one-dimensional tables associated with resources.
- StaticMap – methods for working with two-dimensional "key–value" tables associated with resources.
- Flow – methods for working with finite-state machines associated with resources.
- Mic – methods for implementing *Mandatory Integrity Control* (MIC).

Security event handling by the Kaspersky Security Module

The Kaspersky Security Module calls all methods (rules and expressions) of security models related to an occurring security event. If all rules returned the "granted" result, the security module returns the "granted" decision. If even one rule returned the "denied" result, the security module returns the "denied" decision.

If even one method related to an occurring security event cannot be correctly performed, the security module returns the "denied" decision.

If no rule is related to an occurring security event, the security module returns the "denied" decision. In other words, all interactions between solution components and between those components and the KasperskyOS kernel are denied by default (Default Deny principle) unless those interactions are explicitly allowed by the solution security policy.

PSL language syntax

Basic rules

1. Declarations can be listed in any sequence in a file.
2. One declaration can be written to one or multiple lines.
3. The PSL language is case sensitive.
4. Single-line comments and multi-line comments are supported:

```
/* This is a comment
 * And this, too */
// Another comment
```

Types of declarations

The PSL language has declarations for the following purposes:

- Setting the global parameters of a solution security policy
- Including PSL files in a solution security policy description
- Including EDL files in a solution security policy description
- Create security model objects
- Bind methods of security models to security events
- Creating security audit profiles
- Creating solution security policy tests

Setting the global parameters of a KasperskyOS-based solution security policy

Global parameters include the following parameters of a solution security policy:

- *Execute interface* used by the KasperskyOS kernel when querying the Kaspersky Security Module to notify it about kernel startup or about initiating the startup of a process by the kernel or by other processes. To define this interface, use the following declaration:

```
execute: kl.core.Execute
```

KasperskyOS currently supports only one execute interface (`Execute`) defined in the file named `kl/core/Execute.idl`. (This interface consists of one `main` method, which has no parameters and does not perform any actions. The `main` method is reserved for potential future use.)

- [Optional] Global security audit profile and initial security audit runtime-level. (For more details about profiles and the security audit runtime-level, see Creating security audit profiles.) To define these parameters, use the following declaration:

  ```
  audit default = <security audit profile name> <security audit runtime-level>
  ```

  Example:

  ```
  audit default = global 0
  ```

  The default global profile is the `empty` security audit profile described in the file named `toolchain/include/nk/base.psl` from the KasperskyOS SDK, and the default security audit runtime-level is 0. When the `empty` security audit profile is applied, a security audit is not conducted.

Including PSL files in a KasperskyOS-based solution security policy description

To include a PSL file in a policy description, use the following declaration:

```
use <link to PSL file>._
```

The link to the PSL file is the file path (without the extension and dot before it) relative to the directory that is included in the set of directories where the `nk-psl-gen-c` compiler searches for PSL, IDL, CDL, and EDL files. (This set of directories is defined by the parameters `-I <path to the directory>` when starting the `makekss` script or the `nk-psl-gen-c` compiler.) A dot is used as a separator in a path description. A declaration is ended by the `._` character sequence.

Example:

```
use policy_parts.flow_part._
```

This declaration includes the `flow_part.psl` file, which is located in the `policy_parts` directory. The `policy_parts` directory must reside in one of the directories where the `nk-psl-gen-c` compiler searches for PSL, IDL, CDL, and EDL files. For example, the `policy_parts` directory may reside in the same directory as the PSL file containing this declaration.

Including a PSL file containing a formal description of a security model

To use the methods of a required security model, the policy description must include a PSL file containing a formal description of this model. PSL files containing formal descriptions of security models are located in the KasperskyOS SDK at the following path:

`toolchain/include/nk`

Example:

```
/* Include the base.psl file containing a formal description of the
 * Base security model */
use nk.base._

/* Include the flow.psl file containing a formal description of the
 * Flow security model */
use nk.flow._
/* The nk-psl-gen-c compiler must be configured to search for
 * PSL, IDL, CDL, and EDL files in the toolchain/include directory. */
```

Including EDL files in a KasperskyOS-based solution security policy description

To include an EDL file for the KasperskyOS kernel in a policy description, use any of the following declarations:

```
use EDL kl.core.Core
task class kl.core.Core : kl.core.Core
```

To include an EDL file for a program (such as a driver or application) into a policy description, use one of the following declarations:

```
use EDL <process class name>
task class <process class name> : <link to EDL file>
```

The process class name in the `use EDL` declaration and the link to the EDL file in the `task class` declaration represents the EDL file path (without the extension and dot before it) relative to the directory that is included in the set of directories where the `nk-psl-gen-c` compiler searches for PSL, IDL, CDL, and EDL files. (This set of directories is defined by the `-I <path to the directory>` parameters when starting the `makekss` script or the `nk-psl-gen-c` compiler.) A dot is used as a separator in a path description.

The process class name may differ from the link to the EDL file in the `task class` declaration. The `task class` declaration includes the EDL file in the policy description while ignoring the process class name in this file. The process class name specified in the EDL file is replaced by the process class name specified in the declaration. The same EDL file may be included in the policy description multiple times but with different process class names.

Example:

```
/* Include the UART.edl file that is located in
 * the KasperskyOS SDK at the path sysroot-*-kos/include/kl/drivers. */
use EDL kl.drivers.UART
/* Include the VfsNet.edl file that is located in
 * the KasperskyOS SDK at the path sysroot-*-kos/include/kl.
 * The file is included four times with different process class names. */
task class VfsNet : kl.VfsNet
task class kl.VfsNet : kl.VfsNet
task class kl.VfsNet1 : kl.VfsNet
task class kl.VfsNet2 : kl.VfsNet
/* The nk-psl-gen-c compiler must be configured to search for
 * PSL, IDL, CDL, and EDL files in the sysroot-*-kos/include directory. */
```

In the `task class` declaration, you do not have to specify the link to the EDL file:

```
task class <process class name>
```

This declaration imitates inclusion of an EDL file that contains only the process class name. In other words, if you use this declaration, you don't have to create EDL files containing only the `task class` or `entity` declaration.

Example:

```
/* Imitate the inclusion of an EDL file that contains only
 * the task class start.Empty declaration. */
task class start.Empty
/* Imitate the inclusion of an EDL file that contains only
 * the task class MyClient declaration. */
task class MyClient
```

> The process class names in EDL file inclusion declarations must be unique within the same policy description irrespective of which of the potential EDL file inclusion declarations are being used.

The `nk-psl-gen-c` compiler finds IDL and CDL files via EDL files because EDL files contain the names of components and interfaces. CDL files also contain the names of components and interfaces for searching for IDL and CDL files.

Creating security model objects

To call the methods of a required security model, create an object for this security model.

To create a security model object, use the following declaration:

```
policy object <security model object name> : <security model name> {
    [<security model object parameters>]
}
```

The security model object name must begin with a lowercase letter. The parameters of a security model object are specific to the security model. A description of parameters and examples of creating objects of various security models are provided in the KasperskyOS security models section.

Binding methods of security models to security events

To create an attachment between methods of security models and a security event, use the following declaration:

```
<security event type> [<security event selectors>] {
    [<security audit profile>]
    <called security model methods>
}
```

Security event type

To define the security event type, use the following specifiers:

- `request` – sending IPC requests.

- `response` – sending IPC responses.
- `error` – sending IPC responses containing information about errors.
- `security` – processes querying the Kaspersky Security Module via the security interface.
- `execute` – initializing the startups of processes or the startup of the KasperskyOS kernel.

> When processes interact with the security module, they use a mechanism that is different from IPC. However, when describing a policy, queries sent by processes to the security module can be viewed as the transfer of IPC messages because processes actually transmit messages to the security module (the recipient is not indicated in these messages).
> The IPC mechanism is not used to start processes. However, when the startup of a process is initiated, the kernel queries the security module and provides information about the initiator of the startup and the started process. For this reason, the policy description developer can consider the startup of a process to be analogous to sending an IPC message from the startup initiator to the started process. When the kernel is started, this is analogous to the kernel sending an IPC message to itself.

Security event selectors

Security event selectors let you clarify the description of the defined type of security event. You can use the following selectors:

- `src=<kernel/process class name>` – processes of the defined class or the KasperskyOS kernel are the sources of IPC messages.
- `dst=<kernel/process class name>` – processes of the defined class or the kernel are the recipients of IPC messages.
- `interface=<interface name>` – describes the following security events:
  - Clients attempt to use the endpoints of servers or the kernel with the defined interface.
  - Processes query the Kaspersky Security Module via the defined security interface.
  - The kernel or servers send clients the results from using the endpoints with the defined interface.
- `component=<component name>` – describes the following security events:
  - Clients attempt to use the core or server endpoints provided by the defined component.
  - The kernel or servers send clients the results from using the endpoints provided by the defined component.
- `endpoint=<qualified endpoint name>` – describes the following security events:
  - Clients attempt to use the defined core or server endpoints.
  - The kernel or servers send clients the results from using the defined endpoint.
- `method=<method name>` – describes the following security events:
  - Clients attempt to query servers or the kernel by calling the defined method of the endpoint.

- Processes query the security module by calling the defined method of the security interface.
- The kernel or servers send clients the results from calling the defined method of the endpoint.
- The kernel notifies the security module about its startup by calling the defined method of the execute interface.
- The kernel initiates the startup of processes by calling the defined method of the execute interface.
- Processes initiate the startup of other processes, which results in the kernel calling the defined method of the execute interface.

Process classes, components, instances of components, interfaces, endpoints, and methods must be named the same as they are in the IDL, CDL, and EDL descriptions. The kernel must be named `kl.core.Core`.

The qualified name of an endpoint has the format `<qualified name of component instance>.<endpoint name>`. The qualified name of a component instance has the format `<path to component instance>.<component instance name>`. The path to a component instance is a sequence of component instance names separated by dots. Among these component instances, each subsequent component instance is embedded into the previous one, and the last one contains the component instance with the defined name.

For `security` events, you must specify the qualified name of the security interface method if you need to use the security interface defined in a CDL description. (If you need to use a security interface defined in an EDL description, it is not necessary to specify the qualified name of the method.) The qualified name of a security interface method is a construct in the format `<qualified name of component instance>.<security interface method name>`.

If selectors are not specified, the participants of a security event may be any process and the kernel (except `security` events in which the kernel cannot participate).

You can use combinations of selectors. Selectors can be separated by commas.

There are restrictions on the use of selectors. The `interface`, `component`, and `endpoint` selectors cannot be used for security events of the `execute` type. The `dst`, `component`, and `endpoint` selectors cannot be used for security events of the `security` type.

There are also restrictions on combinations of selectors. For security events of the `request`, `response` and `error` types, the `method` selector can only be used together with one of the `endpoint`, `interface`, or `component` selectors or a combination of them. (The `method`, `endpoint`, `interface` and `component` selectors must be coordinated. In other words, the method, endpoint, interface, and component must be interconnected.) For security events of the `request` type, the `endpoint` selector can be used only together with the `dst` selector. For security events of the `response` and `error` types, the `endpoint` selector can be used only together with the `src` selector.

> The type and selectors of a security event form the security event description. It is recommended to describe security events with maximum precision to allow only the required interactions between different processes and between processes and the kernel. If IPC messages of the same type are always verified when processing the defined event, the description of this event is maximally precise.

To ensure that IPC messages of the same type correspond to a security event description, one of the following conditions must be fulfilled for this description:

- For events of the `request`, `response` and `error` type, the "interface method-endpoint-server class or kernel" chain is unequivocally defined. For example, the security event description `request dst=Server endpoint=net.Net method=Send` corresponds to IPC messages of the same type, and the security event description `request dst=Server` corresponds to any IPC message sent to the `Server`.

- For `security` events, the security interface method is specified.

- The execute-interface method is indicated for `execute` events.

  There is currently support for only one fictitious method of the `main` execute-interface. This method is used by default, so it does not have to be defined through the `method` selector. This way, any description of an `execute` security event corresponds to IPC messages of the same type.

Security audit profile

To define a security audit profile, use the following construct:

```
audit <security audit profile name>
```

If a security audit profile is not defined, the global security audit profile is used.

Called security model methods

To call a security model method, use the following construct:

```
[<security model object name>.]<security model method name> <parameter>
```

Data of PSL-supported types can be used as the parameter. However, the following special considerations should be taken into account:

- If a security model method does not actually have a parameter, this method formally has a Unit-type parameter designated as `()`.

- If a security model method parameter is a dictionary `{<name of field 1> : <value of field 1>[, <name of field 2> : <value of field 2>]...}`, this parameter does not need to be enclosed in parentheses.

- If the security model method parameter is not a dictionary and does not have the Unit type, this parameter must be enclosed in parentheses.

You can call one or more methods by using the same or different security model objects. Security model rules can use the parameter to receive values returned by expressions of security models.

> When a security event is processed by the Kaspersky Security Module, expressions are called before rules. Therefore, expressions do not receive the changes made by rules. For example, if a declaration of attachment between StaticMap security model methods and security events first specifies the `set` rule and then specifies the `get_uncommited` expression for the same resource, the `get_uncommited` expression will return the key value that was defined before the current security event was processed instead of the key value that is defined by the `set` rule when processing the current security event. The key value defined by the `set` rule when processing the current security event can be returned by the `get_uncommited` expression only when processing subsequent security events if the security module returns the "allowed" decision as a result of processing the current security event. If the security module returns a "denied" decision as a result of processing the current security event, all changes made by rules and expressions invoked during processing of the current security event will be discarded.

A security model method (rule or expression) can use the parameter to receive the parameters of interface methods. (For details about obtaining access to parameters of interface methods, see Struct security model.) A security model method can also use the parameter to receive the SID values of processes and the KasperskyOS kernel that are defined by the reserved words `src_sid` and `dst_sid`. The first reserved word refers to the SID of the process (or kernel) that is the source of the IPC message. The second reserved word refers to the SID of the process (or kernel) that is the recipient of the IPC message (`dst_sid` cannot be used for queries to the Kaspersky Security Module).

> You do not have to indicate the security model object name to call certain security model methods. Also, some of the security model methods must be called using operators instead of the call construct. For details about the methods of security models, see KasperskyOS security models.

Embedded constructs for binding methods of security models to security events

In one declaration, you can bind methods of security models to different security events of the same type. To do so, use the match sections that consist of the following types of constructs:

```
match <security event selectors> {
    [<security audit profile>]
    <called security model methods>
}
```

Match sections can be embedded into another match section. A match section simultaneously uses its own security event selectors and the security event selectors at the level of the declaration and all match sections in which this match section is "wrapped". By default, a match section applies the security audit profile of its own container (match section of the preceding level or the declaration level), but you can define a separate security audit profile for the match section.

In one declaration, you can define different variants for processing a security event depending on the conditions in which this event occurred (for example, depending on the state of the finite-state machine associated with the resource). To do so, use the conditional sections that are elements of the following construct:

```
choice <call of the security model expression that verifies fulfillment of conditions> {
    "<condition 1>" : [{] // Conditional section 1
        <called security model methods>
    [}]
    "<condition 2>" : ... // Conditional section 2
...
    _               : ... // Conditional section, if no condition is fulfilled.
}
```

The `choice` construct can be used within a match section. A conditional section uses the security event selectors and security audit profile of its own container.

If multiple conditions described in the `choice` construct are simultaneously fulfilled when a security event is processed, only the one conditional section corresponding to the first true condition on the list is triggered.

> You can verify the fulfillment of conditions in the `choice` construct only by using the expressions that are specially intended for this purpose. Some security models contain these expressions (for more details, see KasperskyOS security models).
> Only text and integer literals, logical values and the `_` character designating an always true condition can be used as conditions.

Examples of binding security model methods to security events

See "Examples of binding security model methods to security events", "Example descriptions of basic security policies for KasperskyOS-based solutions", and "KasperskyOS security models".

Creating security audit profiles

A *security audit* (hereinafter also referred to as an *audit*) is the following sequence of actions. The Kaspersky Security Module provides the KasperskyOS kernel with information about decisions made by this module. Then the kernel forwards this data to the system program `Klog`, which decodes this information and forwards it to the system program `KlogStorage` (data is transmitted via IPC). The latter sends the received audit data to standard output (or standard error) or writes it to a file.

Security audit data (hereinafter referred to as *audit data*) refers to information about decisions made by the Kaspersky Security Module, which includes the actual decisions ("granted" or "denied"), descriptions of security events, results from calling methods of security models, and data on incorrect IPC messages. Data on calls of security model expressions is included in the audit data just like data on calls of security model rules.

To perform a security audit, you need to associate security model objects with security audit profile(s). A *security audit profile* (hereinafter also referred to as an *audit profile*) combines *security audit configurations* (hereinafter also referred to as *audit configurations*), each of which defines the security model objects covered by the audit, and specifies the conditions for conducting the audit. You can define a global audit profile (for more details, see Setting the global parameters of a KasperskyOS-based solution security policy) and/or assign an audit profile(s) at the level of binding security model methods to security events, and/or assign an audit profile(s) at the level of match sections (for more details, see Binding methods of security models to security events).

> Regardless of whether or not audit profiles are being used, audit data contains information about "denied" decisions that were made by the Kaspersky Security Module when IPC messages were invalid and when handling security events that are not associated with any security model rule.

To create a security audit profile, use the following declaration:

```
audit profile <security audit profile name> =
  { <security audit runtime-level 1> :
      // Security audit configuration 1
      { <security model object name 1> :
          { kss: <security audit conditions linked to the results
                  from calls of security model methods>
```

```
        [, <security audit conditions specific to the security model>]
         }
    [, <security model object name 2> :
         {
         ...
         }]
     ...
     }
[, <security audit runtime-level 2> :
     // Security audit configuration 2
     {
     ...
     }]
 ...
 }
```

Security audit runtime-level

The *security audit runtime-level* (hereinafter referred to as the *audit runtime-level*) is a global parameter of a solution security policy and consists of an unsigned integer that defines the active security audit configuration. (The word "runtime-level" here refers to the configuration variant and does not necessarily involve a hierarchy.) The audit runtime-level can be changed during operation of the Kaspersky Security Module. To do so, use a specialized method of the `Base` security model that is called when processes query the security module through the security interface (for more details, see Base security model). The initial audit runtime-level is assigned together with the global audit profile (for more details, see Setting the global parameters of a KasperskyOS-based solution security policy). An `empty` audit profile can be explicitly assigned as the global audit profile.

You can define multiple audit configurations in an audit profile. In different configurations, different security model objects can be covered by the audit and different conditions for conducting the audit can be applied. Audit configurations in a profile correspond to different audit runtime-levels. If a profile does not have an audit configuration corresponding to the current audit runtime-level, the security module will activate the configuration that corresponds to the next-lowest audit runtime-level. If a profile does not have an audit configuration for an audit runtime-level equal to or less than the current level, the security module will not use this profile (in other words, an audit will not be performed for this profile).

The capability to change the audit runtime-level lets you regulate the level of detail of an audit, for example. The higher the audit runtime-level, the higher the level of detail. In other words, a higher audit runtime-level activates audit configurations in which more security model objects are covered by the audit and/or less restrictions are applied in the audit conditions. In addition, you can change the audit runtime-level to switch the audit from one set of logically connected security model objects to another set. For example, a low audit runtime-level activates audit configurations in which the audit covers security model objects related to drivers, a medium audit runtime-level activates audit configurations in which the audit covers security model objects related to the network subsystem, and a high audit runtime-level activates audit configurations in which the audit covers security model objects related to applications.

Name of the security model object

The security model object name is indicated so that the methods provided by this object can be covered by the audit. These methods will be covered by the audit whenever they are called, provided that the conditions for conducting the audit are observed.

> Information about the decisions of the Kaspersky Security Module contained in audit data includes the overall decision of the security module as well as the results from calling individual methods of security modules covered by the audit. To ensure that information about a security module decision is included in audit data, at least one method called during security event handling must be covered by the audit.
> The names of security model objects and the names of methods provided by these objects are included in the audit data.

Security audit conditions

Security audit conditions must be defined separately for each object of a security model.

To define the audit conditions related to the results from calling security model methods, use the following constructs:

- `["granted"]` – the audit is performed if the rules return the "granted" result; the expressions are correctly executed.
- `["denied"]` – the audit is performed if the rules return the "denied" result; the expressions are incorrectly executed.
- `["granted", "denied"]` – the audit is performed regardless of the result returned by rules, and regardless of whether or not rules are correctly fulfilled.
- `[]` – the audit is not performed.

Audit conditions specific to security models are defined by constructs specific to these models (for more details, see KasperskyOS Security models). These conditions apply to rules and expressions. For example, one of these conditions can be the state of a finite-state machine.

Security audit profile for a security audit route

A security audit route includes the kernel and the `Klog` and `KlogStorage` processes, which are connected by IPC channels based on the "kernel – `Klog` – `KlogStorage`" scheme. Security model methods that are associated with transmission of audit data via this route must not be covered by the audit. Otherwise, this will lead to an avalanche of audit data because any data transmission will give rise to new data.

To "suppress" an audit that was defined by a profile with a wider scope (for example, by a global profile or a profile at the level of binding security model methods to a security event), assign an `empty` audit profile at the level of binding security model methods to security events or at the level of the match section.

Examples of security audit profiles

See "Examples of security audit profiles".

Creating and performing tests for a KasperskyOS-based solution security policy

A solution security policy is tested to verify whether or not the policy actually allows what should be allowed and denies what should be denied.

To create a set of tests for a solution security policy, use the following declaration:

```
assert ["<name of test set>"] {
// Constructs in PAL (Policy Assertion Language)
    [setup {<initial part of tests>}]
    sequence ["<test name>"] {<main part of test>}
    ...
    [finally {<final part of tests>}]
}
```

You can create multiple sets of tests by using several of these declarations.

A set of tests can optionally include the initial part of the tests and/or the final part of the tests. The execution of each test from the set begins with whatever is described in the initial part of the test and ends with whatever is described in the final part of the test. This lets you describe the repeated initial and/or final parts of tests in each test.

> After completing each test, all modifications in the Kaspersky Security Module related to the execution of this test are rolled back.

Each test includes one or more test cases.

Test cases

A test case associates a security event description and values of interface method parameters with an expected decision of the Kaspersky Security Module. If the actual security module decision matches the expected decision, the test case passes. Otherwise it fails.

When a test is run, the test cases are executed in the same sequence in which they are described. In other words, the test demonstrates how the security module handles a sequence of security events.

If all test cases within a test pass, the test passes. If even one test case fails to pass, the test fails. A test is terminated on the first failing test case. Each test from the set is run regardless of whether the previous test passed or failed.

A test case description in the PAL language is comprised of the following construct:

```
[<expected decision of security module> ["<test case name>"]] <security event type> <security event selectors>
[{<interface method parameter values>}]
```

The expected decision of the security module can be indicated as `grant` ("granted"), `deny` ("denied") or `any` ("any decision"). You are not required to indicate the expected decision of the security module. The "granted" decision is expected by default. If the `any` value is specified, the security module decision does not have any influence on whether or not the test case passes. In this case, the test case may fail due to errors that occur when the security module processes an IPC message (for example, when the IPC message has an invalid structure).

The name of the test case can be specified if only the expected decision of the security module is specified.

For information about the types and selectors of security events, and about the limitations when using selectors, see Binding methods of security models to security events. Selectors must ensure that the security event description corresponds to IPC messages of the same type. (When security model methods are bound to security events, selectors may not ensure this.)

In security event descriptions, you need to specify the SID instead of the process class name (and the KasperskyOS kernel). However, this requirement does not apply to `execute` events for which the SID of the started process (or kernel) is unknown. To save the SID of the process or kernel to a variable, you need to use the `<-` operator in the test case description in the following format:

```
<variable name> <- execute dst=<kernel/process class name> [<other security event selectors>].
```

The SID value will be assigned to the variable even if startup of the process of the defined class (or kernel) is denied by the tested policy but the "denied" decision is expected.

The PAL language supports abbreviated forms of security event descriptions:

- `security:<Process SID> ! <qualified name of security interface method>` corresponds to `security src=<process SID> method=<qualified name of security interface method>`.
- `request:<client SID> ~> <kernel/server SID> : <qualified name of endpoint.method name>` corresponds to `request src=<client SID> dst=<kernel/server SID> endpoint=<qualified name of endpoint> method=<method name>`.
- `response:<client SID> <~ <kernel/server SID> : <qualified name of endpoint.method name>` corresponds to `response src=<kernel/server SID> dst=<client SID> endpoint=<qualified name of endpoint> method=<method name>`.

The values of interface method parameters must be defined for all types of security events except `execute`. If the values of parameters (and elements of parameters) are not explicitly specified, the default values will be automatically defined. If the interface method has no parameters, specify `{}`. You cannot specify `{}` for security events of the `execute` type.

Interface method parameters and their values must be defined by comma-separated constructs that look as follows:

```
<parameter name> : <value>
```

The names and types of parameters must comply with the IDL description. The sequence order of parameters is not important.

Example definition of parameter values:

```
{ param1 : 23, param2 : "bar", param3 : { collection : [ 5, 7, 12 ], filehandle : { handle : 15, rights : 123 } },
param4 : { name : [ "foo", "baz" ] } }
```

In this example, the number is passed through the `param1` parameter. The string buffer is passed through the `param2` parameter. A structure consisting of two fields is passed through the `param3` parameter. The `collection` field contains an array or sequence of three numeric elements. The `filehandle` field corresponds to the IDL type `Handle` and contains the SID and handle permissions mask. (You do not have to specify the handle permissions mask if, for example, this permissions mask is not checked by the security module. If this is the case, you can specify `filehandle : 15` or `filehandle : { handle : 15 }`, which is interpreted as `filehandle : { handle : 15, rights : 0 }`.) A union or structure containing one field is passed through the `param4` parameter. The `name` field contains an array or sequence of two string buffers.

If the values of parameters (and elements of parameters) are not specified, the system automatically applies the default values corresponding to the IDL types of parameters (and elements of parameters):

- For numerical types: zero
- For the `Handle` type: `{ handle : 0, rights : 0 }`
- A zero-sized byte- or string buffer is the default value for byte- or string buffers.
- The default value for sequences is a sequence with zero elements.
- The default value for arrays is an array of elements with the default values.

- The default value for structures is a structure consisting of fields with the default values.
- For unions, the default value of the first member of the union is applied by default.

Example of applying the default value for a parameter and parameter element:

```
/* Parameter is specified. */
request src=x dst=y endpoint=e method=m { name : { firstname: "a", lastname: "b" } }
/* Parameter is not specified. The applied value of the name parameter will be a structure
 * consisting of two zero-sized string buffers.*/
request src=x dst=y endpoint=e method=m {}
/* Parameter element is not specified. The applied value of the lastname parameter element will be
 * a zero-sized string buffer.*/
request src=x dst=y endpoint=e method=m { name : { firstname: "a"} }
```

Example tests

See "Examples of tests for KasperskyOS-based solution security policies".

Test procedure

The test procedure includes the following steps:

1. Save the tests in one or multiple PSL files (`*.psl` or `*.psl.in`).
2. Add the `CMake` command `add_kss_pal_qemu_tests()` to one of the `CMakeLists.txt` files of the project.

   Use the `PSL_FILES` parameter to define the paths to PSL files containing tests. Use the `DEPENDS` parameter to define the `CMake` targets whose execution will cause the PSL file-dependent IDL, CDL, and EDL files to be put into the directories where the `nk-psl-gen-c` compiler can find them. If `*.psl.in` files are utilized, use the `ENTITIES` parameter to define the names of process classes of system programs. (These system programs are included in a KasperskyOS-based solution that requires security policy testing.)

   Example use of the `CMake` command `add_kss_pal_qemu_tests()` in the file `einit/CMakeLists.txt`:

   ```
   add_kss_pal_qemu_tests (
     PSL_FILES src/security.psl.in
     DEPENDS kos-qemu-image
     ENTITIES ${ENTITIES})
   ```

3. Build and run the tests.

   You must run the Bash build script `cross-build.sh` with the parameter `--target kos-qemu-image-PalTest<N>-sim`, where *N* is the index of the PSL file in the list of PSL files defined through the `PSL_FILES` parameter of the `CMake` command `add_kss_pal_qemu_tests()` at step 2. For example, `--target kos-qemu-image-PalTest0-sim` will create a KasperskyOS-based solution image corresponding to the first PSL file defined through the `PSL_FILES` parameter of the `CMake` command `add_kss_pal_qemu_tests()` and then run that image in QEMU. (Instead of applications and system programs, this solution will contain the program that runs tests.)

   Example:

   ```
   ./cross-build.sh --target kos-qemu-image-PalTest0-sim
   ```

Example test results:

```
[==========] Running 4 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 4 tests from KSS
[ RUN      ] KSS.KssUnitTest_flow_normal
[       OK ] KSS.KssUnitTest_flow_normal (6 ms)
[ RUN      ] KSS.KssUnitTest_flow_ping_must_be_first
/home/work/build/stat/build/install/examples/ping/build/einit/
pal-test/gen_security.psl.test.c:9742: Failure
Expected equality of these values:
  rc
    Which is: -1
  NK_EOK
    Which is: 0
gen_security.psl:116: expect grant

[  FAILED  ] KSS.KssUnitTest_flow_ping_must_be_first (8 ms)
[ RUN      ] KSS.KssUnitTest_flow_ping_ping_is_deny
[       OK ] KSS.KssUnitTest_flow_ping_ping_is_deny (4 ms)
[ RUN      ] KSS.KssUnitTest_flow_test_deny
[       OK ] KSS.KssUnitTest_flow_test_deny (1 ms)
[----------] 4 tests from KSS (29 ms total)

[----------] Global test environment tear-down
[==========] 4 tests from 1 test suite ran. (42 ms total)
[  PASSED  ] 3 tests.
[  FAILED  ] KSS.KssUnitTest_flow_ping_must_be_first (8 ms)
```

The test results contain information about whether or not each test passed or failed. If a test failed, information indicating the location of the description of the failed test example will be displayed in the PSL file.

PSL data types

The data types supported in the PSL language are presented in the table below.

PSL data types

Designations of types	Description of types
`UInt8`, `UInt16`, `UInt32`, `UInt64`	Unsigned integer
`SInt8`, `SInt16`, `SInt32`, `SInt64`	Signed integer
`Boolean`	Boolean type The Boolean type includes two values: `true` and `false`.
`Text`	Text type
`()`	`Unit` type The `Unit` type includes one immutable value. It is used as a stub value in cases when PSL language syntax requires certain data formulation but this data is not actually required. For example, the Unit type can be used to declare a method that does not have any parameters (similar to how the `void` type is used in C/C++).

`"[<text>]"`	Text literal
	A text literal includes one immutable text value.
	Example definitions of text literals:
	`""`
	`"granted"`

`<integer>`	Integer literal
	An integer literal includes one immutable integer value.
	Example definitions of integer literals:
	`12`
	`-5`
	`0xFFFF`

`<type 1> | <type`	Variant type

2> [| <type 3>]…	A variant type combines two or more types and may perform the role of either of them. Examples of definitions of variant types: Boolean | () UInt8 | UInt16 | UInt32 | UInt64 "granted" | "denied"
{ [<name of field 1> : <type of field 1>] [, <name of field 2> : <type of field 2>] … }	Dictionary A dictionary consists of one or more types of fields. A dictionary can be empty. Examples of dictionary definitions: {} { handle : Handle , rights : UInt32 }
[[<type 1>][, <type 2>]…]	Tuple A tuple consists of fields of one or more types in the order in which the types are listed. A tuple can be empty. Examples of tuple definitions: [] ["granted"] [Boolean, Boolean]
Set<<type of elements>>	Set A set includes zero or more unique elements of the same type. Examples of set definitions: Set<"granted" | "denied"> Set<Text>
List<<type of elements>>	List A list includes zero or more elements of the same type. Examples of list definitions: List<Boolean> List<Text | ()>
Map<<key type>, <value type>>	Associative array An associative array includes zero or more entries of the "key-value" type with unique keys. Example of defining an associative array: Map<UInt32, UInt32>
Array<<type of elements>, <number of elements>>	Array An array includes a defined number of elements of the same type. Example of defining an array: Array<UInt8, 42>
Sequence<<type of elements>, <number of elements>>	Sequence A sequence includes from zero to the defined number of elements of the same type. Example of defining a sequence: Sequence<SInt64, 58>

Aliases of certain PSL types

The `nk/base.psl` file from the KasperskyOS SDK defines the data types that are used as the types of parameters (or structural elements of parameters) and returned values for methods of various security models. Aliases and definitions of these types are presented in the table below.

Aliases and definitions of certain data types in PSL

Type alias	Type definition
Unsigned	Unsigned integer UInt8 | UInt16 | UInt32 | UInt64

`Signed`	Signed integer `SInt8 | SInt16 | SInt32 | SInt64`
`Number`	Integer `Unsigned | Signed`
`ScalarLiteral`	Scalar literal `() | Boolean | Number`
`Literal`	Literal `ScalarLiteral | Text`
`Sid`	Type of security ID (SID) `UInt32`
`Handle`	Type of security ID (SID) `Sid`

`HandleDesc`	Dictionary containing fields for the SID and handle permissions mask
	`{ handle : Handle`
	`, rights : UInt32`
	`}`

`Cases`	Type of data received by expressions of security models called in the `choice` construct for verifying fulfillment of conditions `List<Text | ()>`
`KSSAudit`	Type of data defining the conditions for conducting the security audit `Set<"granted" | "denied">`

Mapping IDL types to PSL types

Data types of the IDL language are used to describe the parameters of interface methods. The input data for security model methods have types from the PSL language. The set of data types in the IDL language differs from the set of data types in the PSL language. Parameters of interface methods transmitted in IPC messages can be used as input data for methods of security models, so the policy description developer needs to understand how IDL types are mapped to PSL types.

Integer types of IDL are mapped to integer types of PSL and to variant types of PSL that combine these integer types (including with other types). For example, signed integer types of IDL are mapped to the `Signed` type in PSL, and integer types of IDL are mapped to the `ScalarLiteral` type in PSL.

The `Handle` type in IDL is mapped to the `HandleDesc` type in PSL.

Unions and structures of IDL are mapped to PSL dictionaries.

Arrays and sequences of IDL are mapped to arrays and sequences of PSL, respectively.

String buffers in IDL are mapped to the text type in PSL.

Byte buffers in IDL are not currently mapped to PSL types, so the data contained in byte buffers cannot be used as inputs for security model methods.

Examples of binding security model methods to security events

> Before analyzing examples, you need to become familiar with the Base security model.

Processing the initiation of process startups

```
/* The KasperskyOS kernel and any process
 * in the solution is allowed to start any
 * process. */
execute { grant () }

/* The kernel is allowed to start a process
 * of the Einit class. */
execute src=kl.core.Core, dst=Einit { grant () }

/* An Einit-class process is allowed
 * to start any process in the solution. */
execute src=Einit { grant () }
```

Handling the startup of the KasperskyOS kernel

```
/* The KasperskyOS kernel is allowed to start.
 * (This binding is necessary so that the security
 * module can be notified of the kernel SID. The kernel starts irrespective
 * of whether this is allowed by the solution security policy
 * or denied. If the solution security policy denies the
 * startup of the kernel, after startup the kernel will terminate its
 * execution.) */
execute src=kl.core.Core, dst=kl.core.Core { grant () }
```

Handling IPC request forwarding

```
/* Any client in the solution is allowed to query
 * any server and the KasperskyOS kernel. */
request { grant () }

/* A client of the Client class is allowed to query
 * any server in the solution and the kernel. */
request src=Client { grant () }

/* Any client in the solution is allowed to query
 * a server of the Server class. */
request dst=Server { grant () }

/* A client of the Client class is not allowed to
 * query a server of the Server class. */
request src=Client dst=Server { deny () }

/* A client of the Client class is allowed to
 * query a server of the Server class
 * by calling the Ping method of the net.Net endpoint. */
request src=Client dst=Server endpoint=net.Net method=Ping {
    grant ()
}

/* Any client in the solution is allowed to query
 * a server of the Server class by calling the Send method
 * of the endpoint with the MessExch interface. */
request dst=Server interface=MessExch method=Send {
    grant ()
}
```

Handling IPC response forwarding

```
/* A server of the Server class is allowed to respond to
 * queries of a Client-class client that
 * calls the Ping method of the net.Net endpoint. */
```

```
response src=Server, dst=Client, endpoint=net.Net, method=Ping {
    grant ()
}

/* The server containing the kl.drivers.KIDF component
 * that provide endpoints with the monitor interface is allowed to
 * respond to queries of a DriverManager-class client
 * that uses these endpoints. */
response dst=DriverManager component=kl.drivers.KIDF interface=monitor {
    grant ()
}
```

Handling the transmission of IPC responses containing error information

```
/* A server of the Server class is not allowed to notify a client
 * of the Client class regarding errors that occur
 * when the client queries the server by calling the
 * Ping method of the net.Net endpoint. */
error src=Server, dst=Client, endpoint=net.Net, method=Ping {
    deny ()
}
```

Handling queries sent by processes to the Kaspersky Security Module

```
/* A process of the Sdcard class will receive the
 * "granted" decision from the Kaspersky Security Module
 * by calling the Register method of the security interface.
 * (Using the security interface defined
 * in the EDL description.) */
security src=Sdcard, method=Register {
    grant ()
}

/* A process of the Sdcard class will receive the "denied" decision
 * from the security module when calling the Comp.Register method
 * of the security interface. (Using the security interface
 * defined in the CDL description.) */
security src=Sdcard, method=Comp.Register {
    deny ()
}
```

Using match sections

```
/* A client of the Client class is allowed to query
 * a server of the Server class by calling the Send
 * and Receive methods of the net endpoint. */
request src=Client, dst=Server, endpoint=net {
    match method=Send { grant () }
    match method=Receive { grant () }
}

/* A client of the Client class is allowed to query
 * a server of the Server class by calling the Send
 * and Receive methods of the sn.Net endpoint and the Write and
 * Read methods of the sn.Storage endpoint. */
request src=Client, dst=Server {
    match endpoint=sn.Net {
        match method=Send { grant () }
        match method=Receive { grant () }
    }
    match endpoint=sn.Storage {
        match method=Write { grant () }
        match method=Read { grant () }
    }
}
```

Setting audit profiles

```
/* Set the default global audit profile
 * and initial audit runtime-level of 0 */
audit default = global 0
request src=Client, dst=Server {
    /* Set the parent audit profile at the level of
     * binding methods of security models to
     * security events */
    audit parent
    match endpoint=net.Net, method=Send {
        /* Set a child audit profile at the
         * match section level */
        audit child
        grant ()
    }
    /* This match section applies a
     * parent audit profile. */
    match endpoint=net.Net, method=Receive {
        grant ()
    }
}
/* This binding of the security model method
 * to the security event utilizes the
 * global audit profile. */
response src=Client, dst=Server {
     grant ()
}
```

Example descriptions of basic security policies for KasperskyOS-based solutions

> Before analyzing examples, you need to become familiar with the Struct, Base and Flow security models.

Example 1

The solution security policy in this example allows any interaction between different processes of the `Client`, `Server` and `Einit` classes, and between these processes and the KasperskyOS kernel. The "granted" decision will always be received when these processes query the Kaspersky Security Module. This policy can be used only as a stub during the early stages of development of a KasperskyOS-based solution so that the Kaspersky Security Module does not interfere with interactions. It would be unacceptable to apply such a policy in a real-world KasperskyOS-based solution.

security.psl

```
execute: kl.core.Execute

use nk.base._
use EDL Einit
use EDL Client
use EDL Server
use EDL kl.core.Core

execute { grant () }

request { grant () }

response { grant () }
```

```
error { grant () }

security { grant () }
```

Example 2

The solution security policy in this example imposes limitations on queries sent from clients of the `FsClient` class to servers of the `FsDriver` class. When a client opens a resource controlled by a server of the `FsDriver` class, a finite-state machine in the `unverified` state is associated with this resource. A client of the `FsClient` class is allowed to read data from a resource controlled by a server of the `FsDriver` class only if the finite-state machine associated with this resource is in the `verified` state. To switch a resource-associated finite-state machine from the `unverified` state to the `verified` state, a process of the `FsVerifier` class needs to query the Kaspersky Security Module.

In a real-world KasperskyOS-based solution, this policy cannot be applied because it allows an excessive variety of interactions between different processes and between processes and the KasperskyOS kernel.

security.psl

```
execute: kl.core.Execute

use nk.base._
use nk.flow._
use nk.basic._

policy object file_state : Flow {
    type States = "unverified" | "verified"
    config = {
        states      : ["unverified" , "verified"],
        initial     : "unverified",
        transitions : {
            "unverified" : ["verified"],
            "verified" : []
        }
    }
}


execute { grant () }

request { grant () }

response { grant () }

use EDL kl.core.Core
use EDL Einit
use EDL FsClient
use EDL FsDriver
use EDL FsVerifier

response src=FsDriver, endpoint=operationsComp.operationsImpl, method=Open {
    file_state.init {sid: message.handle.handle}
}

request src=FsClient, dst=FsDriver, endpoint=operationsComp.operationsImpl, method=Read {
    file_state.allow {sid: message.handle.handle, states: ["verified"]}
}

security src=FsVerifier, method=Approve {
    file_state.enter {sid: message.handle.handle, state: "verified"}
}
```

Examples of security audit profiles

Before analyzing examples, you need to become familiar with the Base, Regex and Flow security models.

Example 1

```
// Describing a trace security audit profile
// base – Base security model object
// session – Flow security model object
audit profile trace =
/* If the audit runtime-level is equal to 0, the audit covers
* base object rules when these rules return
* the "denied" result. */
  { 0 :
      { base :
          { kss : ["denied"]
          }
      }
/* If the audit runtime-level is equal to 1, the audit covers methods
* of the session object in the following cases:
* 1.Rules of the session object return any result, and
* the finite-state machine is in a state other than closed.
* 2.A query expression of the session object is executed, and the
* finite-state machine is in a state other than closed. */
  , 1 :
      { session :
          { kss : ["granted", "denied"]
          , omit : ["closed"]
          }
      }
/* If the audit runtime-level is equal to 2, the audit covers methods
* of the session object in the following cases:
* 1.Rules of the session object return any result.
* 2.A query expression of the session object is executed. */
  , 2 :
      { session :
          { kss : ["granted", "denied"]
          }
      }
  }
```

Example 2

```
// Describing a test security audit profile
// base – Base security model object
// re – Regex security model object
audit profile test =
/* If the audit runtime-level is equal to 0, rules of the base object
 * and expressions of the re object are not covered by the audit. */
  { 0 :
      { base :
        { kss : []
        }
      , re :
        { kss : []
        , emit : []
        }
      }
/* If the audit runtime-level is equal to 1, rules of the
 * base object are not covered by the audit, and expressions of the
 * re object are covered by the audit.*/
  , 1 :
      { base :
        { kss : []
        }
      , re :
        { kss : ["granted"]
        , emit : ["match", "select"]
        }
      }
/* If the audit runtime-level is equal to 2, rules of the base object
```

```
 * and expressions of the re object are covered by the audit. Rules
 * of the base object are covered by the audit irrespective of the
 * result that they return.*/
  , 2 :
      { base :
        { kss : ["granted", "denied"]
        }
      , re :
        { kss : ["granted"]
        , emit : ["match", "select"]
        }
      }
  }
```

Examples of tests for KasperskyOS-based solution security policies

> KasperskyOS Community Edition includes the pal_tests example, which demonstrates use of the Policy Assertion Language (PAL) when writing tests for a solution security policy. For more details, refer to pal_tests example.

Example 1

```
/* Test set that includes only one test. */
assert "some tests" {
    /* Test that includes six test cases. */
    sequence "first sequence" {
        /* It is expected that startup of the KasperskyOS kernel is allowed.
         * If this is true, the core variable will be assigned the SID value
         * of the started kernel process. */
        core <- execute dst=kl.core.Core

        /* It is expected that the KasperskyOS kernel is allowed to start a process of the Einit class.
         * If this is true, the einit variable will be assigned the SID value
         * of the started Einit-class process. */
        einit <- execute src=core dst=Einit

        /* It is expected that an Einit-class process is allowed to start a process of the pal_tests.WebServer
class.
         * If this is true, the ws variable will be assigned the SID value
         * of the started pal_tests.WebServer-class process. */
        ws <- execute src=einit dst=pal_tests.WebServer

        /* It is expected that an Einit-class process is allowed to start a process of the pal_tests.UserManager
class.
         * If this is true, the usrmngr variable will be assigned the SID value
         * of the started process of the pal_tests.UserManager class. */
        usrmngr <- execute src=einit dst=pal_tests.UserManager

        /* It is expected that a client of the pal_tests.WebServer-class is not allowed to query
         * a server of the pal_tests.UserManager class by calling the Login method of the auth service
         * if the userName parameter contains an empty string (default value for
         * an unspecified parameter of the string type). */
        deny "Web server calls Login without user name" request src=ws dst=usrmngr endpoint=auth method=Login { }

        /* It is expected that a client of the pal_tests.WebServer-class is allowed to query
         * a server of the pal_tests.UserManager class by calling the Login method of the auth service
         * if the userName parameter is defined. */
        grant "Web server calls Login with user name" request src=ws dst=usrmngr endpoint=auth method=Login {
userName : "Somebody" }
    }
}
```

Example 2

```
/* Test set that includes two tests. */
assert "login tests"{
    /* Initial part of each of the two tests
     * that includes four test cases. */
    setup {
        /* It is expected that startup of the KasperskyOS kernel is allowed.
         * If this is true, the core variable will be assigned the SID value
         * of the started kernel process. */
        core <- execute dst=kl.core.Core

        /* It is expected that the KasperskyOS kernel is allowed to start a process of the Einit class.
         * If this is true, the einit variable will be assigned the SID value
         * of the started Einit-class process. */
        einit <- execute src=core dst=Einit

        /* It is expected that an Einit-class process is allowed to start a process of the pal_tests.WebServer
class.
         * If this is true, the ws variable will be assigned the SID value
         * of the started pal_tests.WebServer-class process. */
        ws <- execute src=einit dst=pal_tests.WebServer

        /* It is expected that an Einit-class process is allowed to start a process of the pal_tests.UserManager
class.
         * If this is true, the usrmngr variable will be assigned the SID value
         * of the started process of the pal_tests.UserManager class. */
        usrmngr <- execute src=einit dst=pal_tests.UserManager
    }
    /* Test that includes seven test cases: four test cases
* in the initial part and three test cases in the main part.*/
    sequence "login after login is denied" {
        /* It is expected that a client of the pal_tests.WebServer-class is allowed to query
         * a server of the pal_tests.UserManager class by calling the Login method of the auth service
         * if the userName parameter is defined. */
        ws ~> usrmngr : auth.Login { userName: "Somebody" }

        /* It is expected that a server of the pal_tests.UserManager class is allowed to respond to
         * a client of the pal_tests.WebServer class if the client calls the Login method of the auth service
         * and this call is allowed. The value of the output parameter "role"
         * is set to "user", which in this case indicates successful authorization of a
         * user with ordinary user permissions. */
        ws <~ usrmngr : auth.Login { role : "user" }

        /* It is expected that a client of the pal_tests.WebServer-class is not allowed to query
         * a server of the pal_tests.UserManager class by calling the Login method of the auth service
         * if successful authorization was completed earlier. */
        deny ws ~> usrmngr : auth.Login { userName: "SomebodyElse" }
    }
    /* Test that includes seven test cases: four test cases
* in the initial part and three test cases in the main part.*/
    sequence "logout after login is granted" {
        /* It is expected that a client of the pal_tests.WebServer-class is allowed to query
         * a server of the pal_tests.UserManager class by calling the Login method of the auth service
         * if the userName parameter is defined. */
        ws ~> usrmngr : auth.Login { userName: "Somebody" }

        /* It is expected that a server of the pal_tests.UserManager class is allowed to respond to
         * a client of the pal_tests.WebServer class if the client calls the Login method of the auth service
         * and this call is allowed. The value of the output parameter "role"
         * is set to "user", which in this case indicates successful authorization of a
         * user with ordinary user permissions. */
        ws <~ usrmngr : auth.Login { role : "user" }

        /* It is expected that a client of the pal_tests.WebServer-class is allowed to query
         * a server of the pal_tests.UserManager class by calling the Logout method of the auth service
         * if successful authorization was completed earlier. */
        ws ~> usrmngr : auth.Logout {}
    }
}
```

Example 3

```
/* Test set that includes only one test. */
assert {
    /* Test that includes ten test cases. */
    sequence {
```

```
        /* It is expected that startup of the KasperskyOS kernel is allowed.
         * If this is true, the core variable will be assigned the SID value
         * of the started kernel process. */
        core <- execute dst=kl.core.Core

        /* It is expected that the KasperskyOS kernel is allowed to start a process of the Einit class.
         * If this is true, the einit variable will be assigned the SID value
         * of the started Einit-class process. */
        einit <- execute src=core dst=Einit

        /* It is expected that an Einit-class process is allowed to start a process of the pal_tests.WebServer
class.
         * If this is true, the ws variable will be assigned the SID value
         * of the started pal_tests.WebServer-class process. */
        ws <- execute src=einit dst=pal_tests.WebServer

        /* It is expected that an Einit-class process is allowed to start a process of the pal_tests.UserManager
class.
         * If this is true, the usrmngr variable will be assigned the SID value
         * of the started process of the pal_tests.UserManager class. */
        usrmngr <- execute src=einit dst=pal_tests.UserManager

        /* It is expected that an Einit-class process is allowed to start a process of the pal_tests.Service
class.
         * If this is true, the srvc variable will be assigned the SID value
         * of the started process of the pal_tests.Service class. */
        srvc <- execute src = einit dst = pal_tests.Service

        /* It is expected that a client of the pal_tests.WebServer-class is not allowed to query
         * a server of the pal_tests.Service class by calling the ProcessRequest method of the handler service
         * if successful authorization was not completed earlier. */
        deny ws ~> srvc : handler.ProcessRequest { }

        /* It is expected that a client of the pal_tests.WebServer-class is allowed to query
         * a server of the pal_tests.UserManager class by calling the Login method of the auth service
         * if the userName parameter is defined. */
        grant request src=ws dst=usrmngr endpoint=auth method=Login { userName : "Somebody" }

        /* It is expected that a server of the pal_tests.UserManager class is allowed to respond to
         * a client of the pal_tests.WebServer class if the client calls the Login method of the auth service
         * and this call is allowed. The value of the output parameter "role"
         * is set to "user", which in this case indicates successful authorization of a
         * user with ordinary user permissions. */
        grant ws <~ usrmngr : auth.Login { role : "user" }

        /* It is expected that a client of the pal_tests.WebServer-class is allowed to query
         * a server of the pal_tests.Service class by calling the ProcessRequest method of the handler service
         * if successful authorization was completed earlier. */
        grant ws ~> srvc : handler.ProcessRequest { }

        /* It is expected that a client of the pal_tests.WebServer-class is not allowed to query
         * a server of the pal_tests.UserManager class by calling the AddUser method of the user service
         * if the user has not been authorized with administrator privileges. */
        deny request src=ws dst=usrmngr endpoint=user method=AddUser { userName : "Anonymous" }
    }
}
```

KasperskyOS security models

Pred security model

Pred security model object

The `basic.psl` file contains a declaration that creates a Pred security model object named `pred`. Consequently, inclusion of the `basic.psl` file into the solution security policy description will create a Pred security model object by default.

A Pred security model object does not have any parameters and cannot be covered by a security audit.

It is not necessary to create additional Pred security model objects.

Pred security model methods

A Pred security model contains expressions that perform comparison operations and return values of the `Boolean` type. To call these expressions, use the following comparison operators:

- `<ScalarLiteral> == <ScalarLiteral>` – "equals".
- `<ScalarLiteral> != <ScalarLiteral>` – "does not equal".
- `<Number> < <Number>` – "is less than".
- `<Number> <= <Number>` – "is less than or equal to".
- `<Number> > <Number>` – "is greater than".
- `<Number> >= <Number>` – "is greater than or equal to".

The Pred security model also contains the `empty` expression that determines whether data contains its own structural elements. This expression returns values of the `Boolean` type. If data does not contain its own structural elements (for example, a set is empty), the expression returns `true`, otherwise it returns `false`. To call the expression, use the following construct:

```
pred.empty (<Text | Set | List | Map>) | ()
```

Bool security model

Bool security model object

The `basic.psl` file contains a declaration that creates a Bool security model object named `bool`. Consequently, inclusion of the `basic.psl` file into the solution security policy description will create a Bool security model object by default.

A Bool security model object does not have any parameters and cannot be covered by a security audit.

It is not necessary to create additional Bool security model objects.

Bool security model methods

The Bool security model contains expressions that perform logical operations and return values of the `Boolean` type. To call these expressions, use the following logical operators:

- `! <Boolean>` – "logical NOT".
- `<Boolean> && <Boolean>` – "logical AND".

- `<Boolean> || <Boolean>` – "logical OR".
- `<Boolean> ==> <Boolean>` – "implication" (`! <Boolean> || <Boolean>`).

The Bool security model also contains the `all`, `any` and `cond` expressions.

The expression `all` performs a "logical AND" for an arbitrary number of values of `Boolean` type. It returns values of the `Boolean` type. It returns `true` if an empty list of values (`[]`) is passed via the parameter. To call the expression, use the following construct:

```
bool.all (<List<Boolean>>)
```

The expression `any` performs a "logical OR" for an arbitrary number of values of `Boolean` type. It returns values of the `Boolean` type. It returns `false` if an empty list of values (`[]`) is passed via the parameter. To call the expression, use the following construct:

```
bool.any (<List<Boolean>>)
```

`cond` expression performs a ternary conditional operation. Returns values of the `ScalarLiteral` type. To call the expression, use the following construct:

```
bool.cond
  { if   : <Boolean> // Condition
  , then : <ScalarLiteral> // Value returned when the condition is true
  , else : <ScalarLiteral> // Value returned when the condition is false
  }
```

In addition to expressions, the Bool security model includes the `assert` rule that works the same as the rule of the same name included in the Base security model.

Math security model

Math security model object

The `basic.psl` file contains a declaration that creates a Math security model object named `math`. Consequently, inclusion of the `basic.psl` file into the solution security policy description will create a Math security model object by default.

A Math security model object does not have any parameters and cannot be covered by a security audit.

It is not necessary to create additional Math security model objects.

Math security model methods

The Math security model contains expressions that perform integer arithmetic operations, bitwise operations, and integer type casting operations. To call a part of these expressions, use the following arithmetic operators:

- `<Number> + <Number>` – "addition". Returns values of the `Number` type.
- `<Number> - <Number>` – "subtraction". Returns values of the `Number` type.

- `<Number> * <Number>` – "multiplication". Returns values of the `Number` type.
- `<Number> | <Number>` – "bitwise OR". Returns values of the `Number` type.
- `<Number> & <Number>` – "bitwise AND". Returns values of the `Number` type.
- `~ <Number>` – "bitwise complement". Returns values of the `Number` type.

The other expressions are as follows:

- `neg (<Signed>)` – "change number sign". Returns values of the `Signed` type.
- `abs (<Signed> )` – "get module of number". Returns values of the `Signed` type.
- `sum (<List<Number>>)` – "add numbers from list". Returns values of the `Number` type. It returns `0` if an empty list of values (`[]`) is passed via the parameter.
- `product (<List<Number>>)` – "multiple numbers from list". Returns values of the `Number` type. It returns `1` if an empty list of values (`[]`) is passed via the parameter.
- `uint64 (<Unsigned>)` – "casting to type `UInt64`".
- `uint32 (<UInt8 | UInt16 | UInt32>)` – "casting to type `UInt32`".
- `uint16 (<UInt8 | UInt16>)` – "casting to type `UInt16`".
- `sint64 (<Signed | UInt8 | UInt16 | UInt32>)` – "casting to type `SInt64`".
- `sint32 (<SInt8 | SInt16 | SInt32 | UInt8 | UInt16>)` – "casting to type `SInt32`".
- `sint16 (<SInt8 | SInt16 | UInt8>)` – "casting to type `SInt16`".
- `signedToUInt64 (<Signed>)` – "casting to type `UInt64`". Negative numbers are converted to positive numbers by subtracting them from *2^64*. For example, *-1* is converted to *2^64-1*.
- `signedToUInt32 (<SInt8 | SInt16 | SInt32>)` – "casting to type `UInt32`". Negative numbers are converted to positive numbers by subtracting them from *2^32*. For example, *-3* is converted to *2^32-3*.
- `signedToUInt16 (<SInt8 | SInt16>)` – "casting to type `UInt16`". Negative numbers are converted to positive numbers by subtracting them from *2^16*. For example, *-5* is converted to *2^16-5*.
- `signedToUInt8 (<SInt8>)` – "casting to type `UInt8`". Negative numbers are converted to positive numbers by subtracting them from *2^8*. For example, *-7* is converted to *2^8-7*.

To call these expressions, use the following construct:

```
math.<expression name> (<parameter>)
```

Struct security model

Struct security model object

The `basic.psl` file contains a declaration that creates a Struct security model object named `struct`. Consequently, inclusion of the `basic.psl` file into the solution security policy description will create a Struct security model object by default.

A Struct security model object does not have any parameters and cannot be covered by a security audit.

It is not necessary to create additional Struct security model objects.

Struct security model methods

The Struct security model contains expressions that provide access to structural data elements. To call these expressions, use the following constructs:

- `<dictionary>.<field name>` – "get access to dictionary field". The type of returned data corresponds to the type of dictionary field.
- `<List | Set | Sequence | Array>.[<element number>]` – "get access to data element". The type of returned data corresponds to the type of elements. The numbering of elements starts with zero. When out of bounds of dataset, the expression terminates with an error and the Kaspersky Security Module returns the "denied" decision.
- `<HandleDesc>.handle` – "get SID". Returns values of the `Handle` type. (For details on the correlation between handles and SID values, see Resource Access Control).
- `<HandleDesc>.rights` – "get handle permissions mask". Returns values of the `UInt32` type.

Parameters of interface methods are saved in a special dictionary named `message`. To obtain access to an interface method parameter, use the following construct:

```
message.<interface method parameter name>
```

The parameter name is specified in accordance with the IDL description.

To obtain access to structural elements of parameters, use the constructs corresponding to expressions of the Struct security model.

> To use expressions of the Struct security model, the security event description must be sufficiently precise so that it corresponds to IPC messages of the same type (for more details, see Binding methods of security models to security events). IPC messages of this type must contain the defined parameters of the interface method, and the interface method parameters must contain the defined structural elements.

Base security model

The Base security model implements basic logic.

A PSL file containing a description of the Base security model is located in the KasperskyOS SDK at the following path:

`toolchain/include/nk/base.psl`

Base security model object

The `base.psl` file contains a declaration that creates a Base security model object named `base`. Consequently, inclusion of the `base.psl` file into the solution security policy description will create a Base security model object by default. Methods of this object can be called without indicating the object name.

A Base security model object does not have any parameters.

A Base security model object can be covered by a security audit. There are no audit conditions specific to the Base security model.

It is necessary to create additional objects of the Base security model in the following cases:

- You need to configure a security audit differently for different objects of the Base security model (for example, you can apply different audit profiles or different audit configurations of the same profile for different objects).
- You need to distinguish between calls of methods provided by different objects of the Base security model (audit data includes the name of the security model method and the name of the object that provides this method, so you can verify that the method of a specific object was called).

Base security model methods

The Base security model contains the following rules:

- `grant ()`

  It has a parameter of the `()` type. It returns the "granted" result.

  Example:

  ```
  /* A client of the foo class is allowed
   * to query a server of the bar class. */
  request src=foo dst=bar { grant () }
  ```

- `assert (<Boolean>)`

  It returns the "granted" result if the `true` value is passed via the parameter. Otherwise it returns the "denied" result.

  Example:

  ```
  /* Any client in the solution will be allowed to query a server of the foo class
   * by calling the Send method of the net.Net endpoint if the port parameter
   * of the Send method will be used to pass a value greater than 80. Otherwise any
   * client in the solution will be prohibited from querying a server of the
   * foo class by calling the Send method of the net.Net endpoint. */
  request dst=foo endpoint=net.Net method=Send { assert (message.port > 80) }
  ```

- `deny (<Boolean>) | ()`

  It returns the "denied" result if the `true` or `()` value is passed via the parameter. Otherwise it returns the "granted" result.

  Example:

  ```
  /* A server of the foo class is not allowed to
   * respond to a client of the bar class. */
  response src=foo dst=bar { deny () }
  ```

- `set_level (<UInt8>)`

  It sets the security audit runtime-level equal to the value passed via this parameter. It returns the "granted" result. (For more details about the security audit runtime-level, see Describing security audit profiles.)

  Example:

```
/* A process of the foo class will receive the "allowed" decision from the
 * Kaspersky Security Module if it calls the
 * SetAuditLevel security interface method to change the security audit runtime-level. */
security src=foo method=SetAuditLevel { set_level (message.audit_level) }
```

Regex security model

The Regex security model implements text data validation based on statically defined regular expressions.

A PSL file containing a description of the Regex security model is located in the KasperskyOS SDK at the following path:

`toolchain/include/nk/regex.psl`

Regex security model object

The `regex.psl` file contains a declaration that creates a Regex security model object named `re`. Consequently, inclusion of the `regex.psl` file into the solution security policy description will create a Regex security model object by default.

A Regex security model object does not have any parameters.

A Regex security model object can be covered by a security audit. In this case, you also need to define the audit conditions specific to the Regex security model. To do so, use the following constructs in the audit configuration description:

- `emit : ["match"]` – the audit is performed if the `match` method is called.
- `emit : ["select"]` – the audit is performed if the `select` method is called.
- `emit : ["match", "select"]` – the audit is performed if the `match` or `select` method is called.
- `emit : []` – the audit is not performed.

It is necessary to create additional objects of the Regex security model in the following cases:

- You need to configure a security audit differently for different objects of the Regex security model (for example, you can apply different audit profiles or different audit configurations of the same profile for different objects).
- You need to distinguish between calls of methods provided by different objects of the Regex security model (audit data includes the name of the security model method and the name of the object that provides this method, so you can verify that the method of a specific object was called).

Regex security model methods

The `Regex` security model contains the following expressions:

- `match {text : <Text>, pattern : <Text>}`

  Returns a value of the `Boolean` type. If the specified `text` matches the `pattern` regular expression, it returns `true`. Otherwise it returns `false`.

  Example:

  ```
  assert (re.match {text : message.text, pattern : "[0-9]*"})
  ```

- `select {text : <Text>}`

  It is intended to be used as an expression that verifies fulfillment of the conditions in the `choice` construct (for details on the `choice` construct, see Binding methods of security models to security events). It checks whether the specified `text` matches regular expressions. Depending on the results of this check, various options for security event handling can be performed.

  Example:

  ```
  choice (re.select {text : "hello world"}) {
      "hello\ .*": grant ()
      ".*world" : grant ()
      _          : deny ()
    }
  ```

Syntax of regular expressions of the Regex security model

A regular expression for the `match` method of the Regex security model can be written in two ways: within the multi-line `regex` block or as a text literal.

> When writing a regular expression as a text literal, all backslash instances must be doubled.

For example, the following two regular expressions are identical:

```
// Regular expression within the multi-line regex block
{ pattern:
    ```regex
    Hello\ world\!
    ```
, text: "Hello world!"
}
// Regular expression as a text literal (doubled backslash)
{ pattern: "Hello\\ world\\!"
, text: "Hello world!"
}
```

Regular expressions for the `select` method of the Regex security model are written as text literals with a double backslash.

A regular expression is defined as a template string and may contain the following:

- Literals (ordinary characters)
- Metacharacters (characters with special meanings)

- White-space characters
- Character sets
- Character groups
- Operators for working with characters

Regular expressions are case sensitive.

Literals and metacharacters in regular expressions

- A literal can be any ASCII character except the metacharacters `.()*&|!?+[]\` and a white-space character. (Unicode characters are not supported.)

  For example, the regular expression `KasperskyOS` corresponds to the text `KasperskyOS`.

- Metacharacters have special meanings that are presented in the table below.

  Special meanings of metacharacters

Metacharacter	Special meaning
`[]`	Square brackets (braces) denote the beginning and end of a set of characters.
`()`	Round brackets (parentheses) denote the beginning and end of a group of characters.
`*`	An asterisk denotes an operator indicating that the character preceding it can repeat zero or more times.
`+`	A plus sign denotes an operator indicating that the character preceding it can repeat one or more times.
`?`	A question mark denotes an operator indicating that the character preceding it can repeat zero or one time.
`!`	An exclamation mark denotes an operator excluding the subsequent character from the list of valid characters.
`|`	A vertical line denotes an operator for selection between characters (logically close to the "OR" conjunction).
`&`	An ampersand denotes an operator for overlapping of multiple conditions (logically close to the "AND" conjunction).
`.`	A dot denotes any character. For example, the regular expression `K.S` corresponds to the sequences of characters `KOS`, `KoS`, `KES` and a multitude of other sequences consisting of three characters that begin with `K` and end with `S`, and in which the second character can be any character: literal, metacharacter, or dot.
`\`	`\<metaSymbol>` A backslash indicates that the metacharacter that follows it will lose its special meaning and instead be interpreted as a literal. A backslash placed before a metacharacter is known as an escape character. For example, a regular expression that consists of a dot metacharacter (`.`) corresponds to any character. However, a regular expression that consists of a backslash with a dot (`\.`) corresponds to only a dot character. Accordingly, a backslash also escapes another subsequent backslash. For example, the regular expression `C:\\Users` corresponds to the sequence of characters `C:\Users`.

- The `^` and `$` characters are not used to designate the start and end of a line.

White-space characters in regular expressions

- A space character has an ASCII code of `20` in a hexadecimal number system and has an ASCII code of `40` in an octal number system. Although a space character does not infer any special meaning, it must be escaped to avoid any ambiguous interpretation by the regular expression interpreter.

  For example, the regular expression `Hello\ world` corresponds to the sequence of characters `Hello world`.

- `\r`

Carriage return character.

- `\n`

  Line break character.

- `\t`

  Horizontal tab character.

Definition of a character based on its octal or hexadecimal code in regular expressions

- `\x{<hex>}`

  Definition of a character using its `hex` code from the ASCII character table. The character code must be less than `0x100`.

  For example, the regular expression `Hello\x{20}world` corresponds to the sequence of characters `Hello world`.

- `\o{<octal>}`

  Definition of a character using its `octal` code from the ASCII character table. The character code must be less than `0o400`.

  For example, the regular expression `\o{75}` corresponds to the `=` character.

Sets of characters in regular expressions

A character set is defined within square brackets `[]` as a list or range of characters. A character set tells the regular expression interpreter that only one of the characters listed in the set or range of characters can be at this specific location in a sequence of characters. A character set cannot be left blank.

- `[<BracketSpec>]` – character set.

  One character corresponds to any character from the `BracketSpec` character set.

  For example, the regular expression `K[OE]S` corresponds to the sequences of characters `KOS` and `KES`.

- `[^<BracketSpec>]` – inverted character set.

  One character corresponds to any character that is not in the `BracketSpec` character set.

  For example, the regular expression `K[^OE]S` corresponds to the sequences of characters `KAS`, `K8S` and any other sequences consisting of three characters that begin with `K` and end with `S`, excluding `KOS` and `KES`.

The `BracketSpec` character set can be listed explicitly or can be defined as a range of characters. When defining a range of characters, the first and last character in the set must be separated with a hyphen.

- `[<Digit1>-<DigitN>]`

  Any number from the range `Digit1`, `Digit2`, … ,`DigitN`.

  For example, the regular expression `[0-9]` corresponds to any numerical digit. The regular expressions `[0-9]` and `[0123456789]` are identical.

  Please note that a range is defined by one character before a hyphen and one character after the hyphen. The regular expression `[1-35]` corresponds only to the characters `1`, `2`, `3` and `5`, and does not represent the range of numbers from `1` to `35`.

- `[<Letter1>-<LetterN>]`

  Any English letter from the range `Letter1`, `Letter2`, … , `LetterN` (these letters must be in the same case).

For example, the regular expression `[a-zA-Z]` corresponds to all letters in uppercase and lowercase from the ASCII character table.

The ASCII code for the upper boundary character of a range must be higher than the ASCII code for the lower boundary character of the range.

For example, the regular expressions `[5-2]` or `[z-a]` are invalid.

The hyphen (minus) `-` character is interpreted as a special character only within a set of characters. Outside of a character set, a hyphen is a literal. For this reason, the `\` metacharacter does not have to precede a hyphen. To use a hyphen as a literal within a character set, it must be indicated first or last in the set.

Examples:

The regular expressions `[-az]` and `[az-]` correspond to the characters `a`, `z` and `-`.

The regular expression `[a-z]` corresponds to any of the 26 English letters from `a` to `z` in lowercase.

The regular expression `[-a-z]` corresponds to any of the 26 English letters from `a` to `z` in lowercase and `-`.

The circumflex (caret character) `^` is interpreted as a special character only within a character set when it is located directly after an opening square bracket. Outside of a character set, a circumflex is a literal. For this reason, the `\` metacharacter does not have to precede a circumflex. To use a circumflex as a literal within a character set, it must be indicated in a location other than first in the set.

Examples:

The regular expression `[0^9]` correspond to the characters `0`, `9` and `^`.

The regular expression `[^09]` corresponds to any character except `0` and `9`.

Within a character set, the metacharacters `*.&|!?+` lose their special meaning and are instead interpreted as literals. Therefore, they do not have to be preceded by the `\` metacharacter. The backslash `\` retains its special meaning within a character set.

For example, the regular expressions `[a.]` and `[a\.]` are identical and correspond to the character `a` and a dot interpreted as a literal.

Groups of characters and operators in regular expressions

A character group uses parentheses `()` to distinguish its portion (subexpression) within a regular expression. Groups are normally used to allocate subexpressions as operands. Groups can be embedded into each other.

> Operators are applied to more than one character in a regular expression only if they are immediately before or after the definition of a set or group of characters. If this is the case, the operator is applied to the entire group or set of characters.

The syntax contains definitions of the following operators (listed in descending order of their priority):

- `!<Expression>`, where `Expression` can be a character, set or group of characters.

  This operator excludes the `Expression` from the list of valid expressions.

Examples:

The regular expression `K!OS` corresponds to the sequences of characters `KoS`, `KES`, and a multitude of other sequences that consist of three characters and begin with `K` and end with `S`, excluding `KOS`.

The regular expression `K!(OS)` corresponds to the sequences of characters `Kos`, `KES`, `KOT`, and a multitude of other sequences that consist of three characters and begin with `K`, excluding `KOS`.

The regular expression `K![OE]S` corresponds to the sequences of characters `KoS`, `KeS`, `K;S`, and a multitude of other sequences that consist of three characters and begin with `K` and end with `S`, excluding `KOS` and `KES`.

- `<Expression>*`, where `Expression` can be a character, set or group of characters.

  This operator means that the `Expression` may occur in the specific position zero or more times.

  Examples:

  The regular expression `0-9*` corresponds to the sequences of characters `0-`, `0-9`, `0-99`, … .

  The regular expression `(0-9)*` corresponds to the empty sequence `""` and the sequences of characters `0-9`, `0-90-9`, … .

  The regular expression `[0-9]*` corresponds to the empty sequence `""` and any non-empty sequence of numbers.

- `<Expression>+`, where `Expression` can be a character, set or group of characters.

  This operator means that the `Expression` may occur in the specific position one or more times.

  Examples:

  The regular expression `0-9+` corresponds to the sequences of characters `0-9`, `0-99`, `0-999`, … .

  The regular expression `(0-9)+` corresponds to the sequences of characters `0-9`, `0-90-9`, … .

  The regular expression `[0-9]+` corresponds to any non-empty sequence of numbers.

- `<Expression>?`, where `Expression` can be a character, set or group of characters.

  This operator means that the `Expression` may occur in the specific position zero or one time.

  Examples:

  The regular expression `https?://` corresponds to the sequences of characters `http://` and `https://`.

  The regular expression `K(aspersky)?OS` corresponds to the sequences of characters `KOS` and `KasperskyOS`.

- `<Expression1><Expression2>` – concatenation. `Expression1` and `Expression2` can be characters, sets or groups of characters.

  This operator does not have a specific designation. In the resulting expression, `Expression2` follows `Expression1`.

  For example, concatenation of the sequences of characters `micro` and `kernel` will result in the sequence of characters `microkernel`.

- `<Expression1>|<Expression2>` – disjunction. `Expression1` and `Expression2` can be characters, sets or groups of characters.

  This operator selects either `Expression1` or `Expression2`.

  Examples:

  The regular expression `KO|ES` corresponds to the sequences of characters `KO` and `ES`, but not `KOS` or `KES` because the concatenation operator has a higher priority than the disjunction operator.

  The regular expression `Press (OK|Cancel)` corresponds to the sequences of characters `Press OK` or `Press Cancel`.

  The regular expression `[0-9]|()` corresponds to numbers from `0` to `9` or an empty string.

- `<Expression1>&<Expression2>` – conjunction. `Expression1` and `Expression2` can be characters, sets or groups of characters.

  This operator intersects the result of `Expression1` with the result of `Expression2`.

  Examples:

  The regular expression `[0-9]&[^3]` corresponds to numbers from `0` to `9`, excluding `3`.

  The regular expression `[a-zA-Z]&()` corresponds to all English letters and an empty string.

HashSet security model

The HashSet security model associates resources with one-dimensional tables of unique values of the same type, adds or deletes these values, and checks whether a defined value is in the table. For example, a process of the network server can be associated with the set of ports that this server is allowed to open. This association can be used to check whether the server is allowed to initiate the opening of a port.

A PSL file containing a description of the HashSet security model is located in the KasperskyOS SDK at the following path:

`toolchain/include/nk/hashmap.psl`

HashSet security model object

To use the HashSet security model, you need to create an object or objects of this model.

A HashSet security model object contains a pool of one-dimensional tables of the same size intended for storing the values of one type. A resource can be associated with only one table from the tables pool of each HashSet security model object.

A HashSet security model object has the following parameters:

- `type Entry` – type of values in tables (these can be integer types, `Boolean` type, and dictionaries and tuples based on integer types and the `Boolean` type).

- `config` – configuration of the pool of tables:

  - `set_size` – size of the table.

  - `pool_size` – number of tables in the pool.

All parameters of a HashSet security model object are required.

Example:

```
policy object s : HashSet {
    type Entry = UInt32

    config =
      { set_size : 5
      , pool_size : 2
      }
}
```

A HashSet security model object can be covered by a security audit. There are no audit conditions specific to the HashSet security model.

It is necessary to create multiple objects of the HashSet security model in the following cases:

- You need to configure a security audit differently for different objects of the HashSet security model (for example, you can apply different audit profiles or different audit configurations of the same profile for different objects).
- You need to distinguish between calls of methods provided by different objects of the HashSet security model (audit data includes the name of the security model method and the name of the object that provides this method, so you can verify that the method of a specific object was called).
- You need to use tables of different sizes and/or with different types of values.

HashSet security model init rule

```
init {sid : <Sid>}
```

It associates a free table from the tables pool with the `sid` resource. If the free table contains values after its previous use, these values are deleted.

It returns the "allowed" result if an association was created between the table and the `sid` resource.

It returns the "denied" result in the following cases:

- There are no free tables in the pool.
- The `sid` resource is already associated with a table from the tables pool of the HashSet security model object being used.
- The `sid` value is outside of the permissible range.

Example:

```
/* A process of the Server class will be allowed to start if,
 * at startup initiation, an association will be created
 * between this process and the table. Otherwise the startup of a process of the
 * Server class will be denied. */
execute dst=Server {
    s.init {sid : dst_sid}
}
```

HashSet security model fini rule

```
fini {sid : <Sid>}
```

It deletes the association between the table and the `sid` resource (the table becomes free).

It returns the "allowed" result if the association between the table and the `sid` resource was deleted.

It returns the "denied" result in the following cases:

- The `sid` resource is not associated with a table from the tables pool of the HashSet security model object being used.
- The `sid` value is outside of the permissible range.

HashSet security model add rule

```
add {sid : <Sid>, entry : <Entry>}
```

It adds the `entry` value to the table associated with the `sid` resource.

It returns the "allowed" result in the following cases:

- The rule added the `entry` value to the table associated with the `sid` resource.
- The table associated with the `sid` resource already contains the `entry` value.

It returns the "denied" result in the following cases:

- The table associated with the `sid` resource is completely full.
- The `sid` resource is not associated with a table from the tables pool of the HashSet security model object being used.
- The `sid` value is outside of the permissible range.

Example:

```
/* A process of the Server class will receive the "allowed" decision from
 * the Kaspersky Security Module by calling the
 * Add security interface method if, when this method is called, the value
 * 5 will be added to the table associated with this
 * process, or is already in the table. Otherwise
 * a process of the Server class will receive the "denied" decision from the
 * security module by calling the
 * Add security interface method. */
security src=Server, method=Add {
    s.add {sid : src_sid, entry : 5}
}
```

HashSet security model remove rule

```
remove {sid : <Sid>, entry : <Entry>}
```

It deletes the `entry` value from the table associated with the `sid` resource.

It returns the "allowed" result in the following cases:

- The rule deleted the `entry` value from the table associated with the `sid` resource.

- The table associated with the `sid` resource does not contain the `entry` value.

It returns the "denied" result in the following cases:

- The `sid` resource is not associated with a table from the tables pool of the HashSet security model object being used.
- The `sid` value is outside of the permissible range.

HashSet security model contains expression

```
contains {sid : <Sid>, entry : <Entry>}
```

It checks whether the `entry` value is in the table associated with the `sid` resource.

It returns a value of the `Boolean` type. If the `entry` value is in the table associated with the `sid` resource, it returns `true`. Otherwise it returns `false`.

It runs incorrectly in the following cases:

- The `sid` resource is not associated with a table from the tables pool of the HashSet security model object being used.
- The `sid` value is outside of the permissible range.

When the expression runs incorrectly, the Kaspersky Security Module returns the "denied" decision.

Example:

```
/* A process of the Server class will receive the "allowed" decision from
 * the Kaspersky Security Module by calling the
 * Check security interface method if the value 42 is in the table
 * associated with this process. Otherwise a process of the
 * Server class will receive the "denied" decision from the security module
 * by calling the Check security interface method. */
security src=Server, method=Check {
    assert(s.contains {sid : src_sid, entry : 42})
}
```

StaticMap security model

The StaticMap security model associates resources with two-dimensional "key–value" tables, reads and modifies the values of keys. For example, a process of the driver can be associated with the MMIO memory region that this driver is allowed to use. This will require two keys whose values define the base address and the size of the MMIO memory region. This association can be used to check whether the driver can query the MMIO memory region that it is attempting to access.

Keys in the table have the same type but are unique and immutable. The values of keys in the table have the same type.

There are two simultaneous instances of the table: base instance and working instance. Both instances are initialized by the same data. Changes are made first to the working instance and then can be added to the base instance, or vice versa: the working instance can be changed by using previous values from the base instance. The values of keys can be read from the base instance or working instance of the table.

A PSL file containing a description of the StaticMap security model is located in the KasperskyOS SDK at the following path:

`toolchain/include/nk/staticmap.psl`

StaticMap security model object

To use the StaticMap security model, you need to create an object or objects of this model.

A StaticMap security model object contains a pool of two-dimensional "key–value" tables that have the same size. A resource can be associated with only one table from the tables pool of each StaticMap security model object.

A StaticMap security model object has the following parameters:

- `type Value` – type of values of keys in tables (integer types are supported).
- `config` – configuration of the pool of tables:
  - `keys` – table containing keys and their default values (keys have the `Key = Text | List<UInt8>` type).
  - `pool_size` – number of tables in the pool.

All parameters of a StaticMap security model object are required.

Example:

```
policy object m : StaticMap {
    type Value = UInt16

    config =
      { keys:
        { "k1" : 0
        , "k2" : 1
        }
      , pool_size : 2
      }
}
```

A StaticMap security model object can be covered by a security audit. There are no audit conditions specific to the StaticMap security model.

It is necessary to create multiple objects of the StaticMap security model in the following cases:

- You need to configure a security audit differently for different objects of the StaticMap security model (for example, you can apply different audit profiles or different audit configurations of the same profile for different objects).
- You need to distinguish between calls of methods provided by different objects of the StaticMap security model (audit data includes the name of the security model method and the name of the object that provides this method, so you can verify that the method of a specific object was called).

- You need to use tables with different sets of keys and/or different types of key values.

StaticMap security model init rule

```
init {sid : <Sid>}
```

It associates a free table from the tables pool with the `sid` resource. Keys are initialized by the default values.

It returns the "allowed" result if an association was created between the table and the `sid` resource.

It returns the "denied" result in the following cases:

- There are no free tables in the pool.
- The `sid` resource is already associated with a table from the tables pool of the StaticMap security model object being used.
- The `sid` value is outside of the permissible range.

Example:

```
/* A process of the Server class will be allowed to start if,
 * at startup initiation, an association will be created
 * between this process and the table. Otherwise the startup of a process of the
 * Server class will be denied. */
execute dst=Server {
    m.init {sid : dst_sid}
}
```

StaticMap security model fini rule

```
fini {sid : <Sid>}
```

It deletes the association between the table and the `sid` resource (the table becomes free).

It returns the "allowed" result if the association between the table and the `sid` resource was deleted.

It returns the "denied" result in the following cases:

- The `sid` resource is not associated with a table from the tables pool of the StaticMap security model object being used.
- The `sid` value is outside of the permissible range.

StaticMap security model set rule

```
set {sid : <Sid>, key : <Key>, value : <Value>}
```

It assigns the specified `value` to the specified `key` in the working instance of the table associated with the `sid` resource.

It returns the "allowed" result if the specified `value` was assigned to the specified `key` in the working instance of the table associated with the `sid` resource. (The current value of the key will be overwritten even if it is equal to the new value.)

It returns the "denied" result in the following cases:

- The specified `key` is not in the table associated with the `sid` resource.
- The `sid` resource is not associated with a table from the tables pool of the StaticMap security model object being used.
- The `sid` value is outside of the permissible range.

Example:

```
/* A process of the Server class will receive the "allowed" decision from
 * the Kaspersky Security Module by calling the
 * Set security interface method if, when this method is called, the value 2
 * will be assigned to key k1 in the working instance of the table
 * associated with this process. Otherwise a process of the
 * Server class will receive the "denied" decision from the security module
 * by calling the Set security interface method. */
security src=Server, method=Set {
    m.set {sid : src_sid, key : "k1", value : 2}
}
```

StaticMap security model commit rule

```
commit {sid : <Sid>}
```

It copies the values of keys from the working instance to the base instance of the table associated with the `sid` resource.

It returns the "allowed" result if the values of keys were copied from the working instance to the base instance of the table associated with the `sid` resource.

It returns the "denied" result in the following cases:

- The `sid` resource is not associated with a table from the tables pool of the StaticMap security model object being used.
- The `sid` value is outside of the permissible range.

StaticMap security model rollback rule

```
rollback {sid : <Sid>}
```

It copies the values of keys from the base instance to the working instance of the table associated with the `sid` resource.

It returns the "allowed" result if the values of keys were copied from the base instance to the working instance of the table associated with the `sid` resource.

It returns the "denied" result in the following cases:

- The `sid` resource is not associated with a table from the tables pool of the StaticMap security model object being used.
- The `sid` value is outside of the permissible range.

StaticMap security model get expression

```
get {sid : <Sid>, key : <Key>}
```

It returns the value of the specified `key` from the base instance of the table associated with the `sid` resource.

It returns a value of the `Value` type.

It runs incorrectly in the following cases:

- The specified `key` is not in the table associated with the `sid` resource.
- The `sid` resource is not associated with a table from the tables pool of the StaticMap security model object being used.
- The `sid` value is outside of the permissible range.

When the expression runs incorrectly, the Kaspersky Security Module returns the "denied" decision.

Example:

```
/* A process of the Server class will receive the "allowed" decision from
 * the Kaspersky Security Module by calling the
 * Get security interface method if the value of key k1 in the base
 * instance of the table associated with this process
 * is not zero. Otherwise a process of the Server class will receive
 * the "denied" decision from the security module
 * by calling the Get security interface method. */
security src=Server, method=Get {
    assert(m.get {sid : src_sid, key : "k1"} != 0)
}
```

StaticMap security model get_uncommitted expression

```
get_uncommitted {sid: <Sid>, key: <Key>}
```

It returns the value of the specified `key` from the working instance of the table associated with the `sid` resource.

It returns a value of the `Value` type.

It runs incorrectly in the following cases:

- The specified `key` is not in the table associated with the `sid` resource.
- The `sid` resource is not associated with a table from the tables pool of the StaticMap security model object being used.
- The `sid` value is outside of the permissible range.

When the expression runs incorrectly, the Kaspersky Security Module returns the "denied" decision.

Flow security model

The Flow security model associates resources with finite-state machines, receives and modifies the states of finite-state machines, and checks whether the state of the finite-state machine is within the defined set of states. For example, a process can be associated with a finite-state machine to allow or prohibit this process from using storage and/or the network depending on the state of the finite-state machine.

A PSL file containing a description of the Flow security model is located in the KasperskyOS SDK at the following path:

`toolchain/include/nk/flow.psl`

Flow security model object

To use the Flow security model, you need to create an object or objects of this model.

One Flow security model object associates a set of resources with a set of finite-state machines that have the same configuration. A resource can be associated with only one finite-state machine of each Flow security model object.

A Flow security model object has the following parameters:

- `type State` – type that determines the set of states of the finite-state machine (variant type that combines text literals).
- `config` – configuration of the finite-state machine:
  - `states` – set of states of the finite-state machine (must match the set of states defined by the `State` type).
  - `initial` – initial state of the finite-state machine.
  - `transitions` – description of the permissible transitions between states of the finite-state machine.

All parameters of a Flow security model object are required.

Example:

```
policy object service_flow : Flow {
    type State = "sleep" | "started" | "stopped" | "finished"

    config = { states      : ["sleep", "started", "stopped", "finished"]
             , initial     : "sleep"
             , transitions : { "sleep"   : ["started"]
                             , "started" : ["stopped", "finished"]
```

```
                                , "stopped" : ["started", "finished"]
                                }
                }
}
```

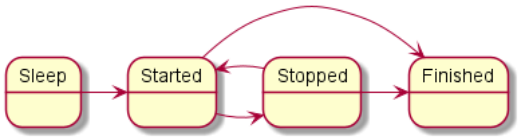


Diagram of finite-state machine states in the example

A Flow security model object can be covered by a security audit. You can also define the audit conditions specific to the Flow security model. To do so, use the following construct in the audit configuration description:

`omit : [<"state 1">[, ...]]` – the audit is not performed if the finite-state machine is in one of the listed states.

It is necessary to create multiple objects of the Flow security model in the following cases:

- You need to configure a security audit differently for different objects of the Flow security model (for example, you can apply different audit profiles or different audit configurations of the same profile for different objects).
- You need to distinguish between calls of methods provided by different objects of the Flow security model (audit data includes the name of the security model method and the name of the object that provides this method, so you can verify that the method of a specific object was called).
- You need to use finite-state machines with different configurations.

Flow security model init rule

```
init {sid : <Sid>}
```

It creates a finite-state machine and associates it with the `sid` resource. The created finite-state machine has the configuration defined in the settings of the Flow security model object being used.

It returns the "granted" result if an association was created between the finite-state machine and the `sid` resource.

It returns the "denied" result in the following cases:

- The `sid` resource is already associated with a finite-state machine of the Flow security model object being used.
- The `sid` value is outside of the permissible range.

Example:

```
/* A process of the Server class will be allowed to start
 * if, at startup initiation, an association will be created
 * between this process and the finite-state machine.
 * Otherwise the startup of the Server-class process will be denied. */
execute dst=Server {
    service_flow.init {sid : dst_sid}
}
```

Flow security model fini rule

```
fini {sid : <Sid>}
```

It deletes the association between the finite-state machine and the `sid` resource. The finite-state machine that is no longer associated with the resource is destroyed.

It returns the "granted" result if the association between the finite-state machine and the `sid` resource was deleted.

It returns the "denied" result in the following cases:

- The `sid` resource is not associated with a finite-state machine of the Flow security model object being used.
- The `sid` value is outside of the permissible range.

Flow security model enter rule

```
enter {sid : <Sid>, state : <State>}
```

It switches the finite-state machine associated with the `sid` resource to the specified `state`.

It returns the "granted" result if the finite-state machine associated with the `sid` resource was switched to the specified `state`.

It returns the "denied" result in the following cases:

- The transition to the specified `state` from the current state is not permitted by the configuration of the finite-state machine associated with the `sid` resource.
- The `sid` resource is not associated with a finite-state machine of the Flow security model object being used.
- The `sid` value is outside of the permissible range.

Example:

```
/* Any client in the solution will be allowed to query
 * a server of the Server class if the finite-state machine
 * associated with this server will be switched to
 * the "started" state when initiating the query. Otherwise
 * any client in the solution will be denied to query
 * a server of the Server class. */
request dst=Server {
    service_flow.enter {sid : dst_sid, state : "started"}
}
```

Flow security model allow rule

```
allow {sid : <Sid>, states : <Set<State>>}
```

It verifies that the state of the finite-state machine associated with the `sid` is in the set of defined `states`.

It returns the "granted" result if the state of the finite-state machine associated with the `sid` resource is in the set of defined `states`.

It returns the "denied" result in the following cases:

- The state of the finite-state machine associated with the `sid` resource is not in the set of defined `states`.
- The `sid` resource is not associated with a finite-state machine of the Flow security model object being used.
- The `sid` value is outside of the permissible range.

Example:

```
/* Any client in the solution is allowed to query a server
 * of the Server class if the finite-state machine associated with this server
 * is in the started or stopped state. Otherwise any client
 * in the solution will be prohibited from querying a server of the Server class. */
request dst=Server {
    service_flow.allow {sid : dst_sid, states : ["started", "stopped"]}
}
```

Flow security model query expression

```
query {sid : <Sid>}
```

It is intended to be used as an expression that verifies fulfillment of the conditions in the `choice` construct (for details on the `choice` construct, see Binding methods of security models to security events). It checks the state of the finite-state machine associated with the `sid` resource. Depending on the results of this check, various options for security event handling can be performed.

It runs incorrectly in the following cases:

- The `sid` resource is not associated with a finite-state machine of the Flow security model object being used.
- The `sid` value is outside of the permissible range.

When the expression runs incorrectly, the Kaspersky Security Module returns the "denied" decision.

Example:

```
/* Any client in the solution is allowed to query
 * a server of the ResourceDriver class if the finite-state machine
 * associated with this server is in the
 * "started" or "stopped" state. Otherwise any client in the solution
 * is prohibited from querying a server of the ResourceDriver class. */
request dst=ResourceDriver {
    choice (service_flow.query {sid : dst_sid}) {
        "started"   : grant ()
        "stopped"   : grant ()
        _           : deny ()
    }
}
```

Mic security model

The Mic security model implements mandatory integrity control. In other words, this security model provides the capability to manage data streams between different processes and between processes and the KasperskyOS kernel by controlling the integrity levels of processes, the kernel, and resources that are used via IPC.

> In Mic security model terminology, processes and the kernel are called subjects while resources are called objects. However, the information provided in this section slightly deviates from the terminology of the Mic security model. In this section, the term "object" is not used to refer to a "resource".

Data streams are generated between subjects when the subjects interact via IPC.

The *integrity level of a subject/resource* is the level of trust afforded to the subject/resource. The degree of trust in a subject depends on whether the subject interacts with untrusted external software/hardware systems or whether the subject has a proven quality level, for example. (The kernel has a high level of integrity.) The degree of trust in a resource depends on whether this resource was created by a trusted subject within a software/hardware system running KasperskyOS or if it was received from an untrusted external software/hardware system, for example.

The Mic security model is characterized by the following provisions:

- By default, data streams from subjects with less integrity to subjects with higher integrity are prohibited. You have the option of permitting such data streams if you can guarantee that the subjects with higher integrity will not be compromised.
- A resource consumer is prohibited from writing data to a resource if the integrity level of the resource is higher than the integrity level of the resource consumer.
- By default, a resource consumer is prohibited from reading data from a resource if the integrity level of the resource is lower than the integrity level of the resource consumer. You have the option to allow the resource consumer to perform such an operation if you can guarantee that the resource consumer will not be compromised.

Methods of a Mic security model let you perform the following operations:

- Assign integrity levels to subjects and resources.
- Unassign the integrity level from resources.
- Verify the permissibility of data streams based on a comparison of integrity levels.
- Increase the integrity levels of resources.

A PSL file containing a description of the Mic security model is located in the KasperskyOS SDK at the following path:

`toolchain/include/nk/mic.psl`

For an example of using the Mic security model, we can examine a secure software update for a software/hardware system running KasperskyOS. Four processes are involved in the update:

- `Downloader` is a low-integrity process that downloads a low-integrity update image from a remote server on the Internet.
- `Verifier` is a high-integrity process that verifies the digital signature of the low-integrity update image (high-integrity process that can read data from a low-integrity resource).
- `FileSystem` is a high-integrity process that manages the file system.
- `Updater` is a high-integrity process that applies an update.

A software update is performed according to the following scenario:

1. The `Downloader` downloads an update image and saves it to a file by transferring the contents of the image to the `FileSystem`. A low integrity level is assigned to this file.
2. The `Verifier` receives the update image from the `FileSystem` by reading the high-integrity file, and verifies its digital signature. If the signature is correct, the `Verifier` queries the `FileSystem` so that the `FileSystem` creates a copy of the file containing the update image. A high integrity level is assigned to the new file.
3. The `Updater` receives the update image from the `FileSystem` by reading the high-integrity file, and applies the update.

In this example, the Mic security model ensures that the high-integrity `Updater` process can read data only from a high-integrity update image. As a result, the update can be applied only after the digital signature of the update image is verified.

Mic security model object

To use the Mic security model, you need to create an object or objects of this model. You also need to assign a set of integrity levels for subjects and resources.

A Mic security model object has the following parameters:

- `config` – set of integrity levels or configuration of a set of integrity levels:
  - `degrees` – set of gradations for generating a set of integrity levels.
  - `categories` – set of categories for generating a set of integrity levels.

Examples:

```
policy object mic : Mic {
    config = ["LOW", "MEDIUM", "HIGH"]
}

policy object mic_po : Mic {
    config =
      { degrees    : ["low", "high"]
      , categories : ["net", "log"]
      }
}
```

A set of integrity levels is a partially ordered set that is linearly ordered or contains incomparable elements. The set {LOW, MEDIUM, HIGH} is linearly ordered because all of its elements are comparable to each other. Incomparable elements arise when a set of integrity levels is defined through a set of gradations and a set of categories. In this case, the set of integrity levels *L* is a Cartesian product of the Boolean set of categories *C* multiplied by the set of gradations *D*:

$$L = 2^C \times D.$$

The `degrees` and `categories` parameters in this example define the following set:

{

{}/low, {}/high,

{net}/low, {net}/high,

{log}/low, {log}/high,

{net,log}/low, {net,log}/high

}

In this set, {} means an empty set.

The order relation between elements of the set of integrity levels *L* is defined as follows:

$$l_i = A/B,$$
$$l_j = E/F,$$
$$l_i \prec l_j \Leftrightarrow \begin{cases} A \subseteq E, \\ B \leq F. \end{cases}$$

According to this order relation, the *j*th element exceeds the *i*th element if the subset of categories *E* includes the subset of categories *A*, and gradation *F* is greater than or equal to gradation *B*. Examples of comparing elements of the set of integrity levels *L*:

- The {net,log}/high element exceeds the {log}/low element because the "high" gradation is greater than the "low" gradation, and the subset of categories {net,log} includes the subset of categories {log}.
- The {net,log}/low element exceeds the {log}/low element because the levels of gradations for these elements are equal, and the subset of categories {net,log} includes the subset of categories {log}.
- The {net,log}/high element is the highest because it exceeds all other elements.
- The {}/low element is the lowest because all other elements exceed this element.
- The {net}/low and {log}/high elements are incomparable because the "high" gradation is greater than the "low" gradation but the subset of categories {log} does not include the subset of categories {net}.
- The {net,log}/low and {log}/high elements are incomparable because the "high" gradation is greater than the "low" gradation but the subset of categories {log} does not include the subset of categories {net,log}.

> For subjects and resources that have incomparable integrity levels, the Mic security model provides conditions that are analogous to the conditions that the security model provides for subjects and resources that have comparable integrity levels.

> By default, data streams between subjects that have incomparable integrity levels are prohibited. However, you have the option to allow such data streams if you can guarantee that the subjects receiving data will not be compromised. A resource consumer is prohibited from writing data to a resource and read data from a resource if the integrity level of the resource is incomparable to the integrity level of the resource consumer. You have the option to allow the resource consumer to read data from a resource if you can guarantee that the resource consumer will not be compromised.

A Mic security model object can be covered by a security audit. There are no audit conditions specific to the Mic security model.

It is necessary to create multiple objects of the Mic security model in the following cases:

- You need to configure a security audit differently for different objects of the Mic security model (for example, you can apply different audit profiles or different audit configurations of the same profile for different objects).
- You need to distinguish between calls of methods provided by different objects of the Mic security model (audit data includes the name of the security model method and the name of the object that provides this method, so you can verify that the method of a specific object was called).
- You need to use multiple variants of mandatory integrity control that may have different sets of integrity levels for subjects and resources, for example.

Mic security model create rule

```
create { source    : <Sid>
       , target    : <Sid>
       , container : <Sid | ()>
       , driver    : <Sid>
       , level     : <Level | ... | ()>
       }
```

Assign the specified integrity `level` to the `target` resource in the following situation:

- The `source` process initiates creation of the `target` resource.
- The `target` resource is managed by the `driver` subject, which is the resource provider or the KasperskyOS kernel.
- The `container` resource is a container for the `target` resource (for example, a directory is a container for files and/or other directories).

If the `container` field has the value `()`, the `target` resource is considered to be the root resource, which means that it has no container.

To define the integrity `level`, values of the `Level` type are used:

```
type Level = LevelFull | LevelNoCategory

type LevelFull =
  { degree     : Text | ()
  , categories : List<Text> | ()
  }

type LevelNoCategory = Text
```

The rule returns the "granted" result if a specific integrity `level` was assigned to the `target` resource.

The rule returns the "denied" result in the following cases:

- The `level` value exceeds the integrity level of the `source` process, `driver` subject or `container` resource.
- The `level` value is incomparable to the integrity level of the `source` process, `driver` subject or `container` resource.
- An integrity level was not assigned to the `source` process, `driver` subject, or `container` resource.
- The value of `source`, `target`, `container` or `driver` is outside of the permissible range.

Example:

```
/* A server of the updater.Realmserv class will be allowed to respond to
 * queries of any client in the solution calling the resolve method
 * of the realm.Reader endpoint if the resource whose creation is requested
 * by the client will be assigned the LOW integrity level during response initiation.
 * Otherwise a server of the updater.Realmserv class will be prohibited from responding to
 * queries of any client calling the resolve method of the realm.Reader endpoint. */
response src=updater.Realmserv,
         endpoint=realm.Reader {
             match method=resolve {
                 mic.create { source : dst_sid
                            , target : message.handle.handle
                            , container : ()
                            , driver : src_sid
                            , level : "LOW"
                            }
             }
}
```

Mic security model delete rule

```
delete { source    : <Sid>
       , target    : <Sid>
       , container : <Sid | ()>
       , driver    : <Sid>
       }
```

Unassigns the integrity level from the `target` resource in the following situation:

- The `source` process initiates deletion of the `target` resource.
- The `target` resource is managed by the `driver` subject, which is the resource provider or the KasperskyOS kernel.
- The `container` resource is a container for the `target` resource (for example, a directory is a container for files and/or other directories).

If the `container` field has the value `()`, the `target` resource is considered to be the root resource, which means that it has no container.

The rule returns the "granted" result if it unassigned the integrity level from the `target` resource.

The rule returns the "denied" result in the following cases:

- The integrity level of the `target` resource exceeds the integrity level of the `source` process or `driver` subject.
- The integrity level of the `target` resource is incomparable to the integrity level of the `source` process or `driver` subject.
- An integrity level was not assigned to the `source` process, `driver` subject, `target` resource or `container` resource.
- The value of `source`, `target`, `container` or `driver` is outside of the permissible range.

Example:

```
/* Any client in the solution will be allowed to query a server of the foo class
 * updater.Realmserv class by calling the del method of the realm.Reader endpoint if the
 * integrity level will be unassigned from the resource whose deletion is requested by the client.
 * Otherwise, any client in the solution will be prohibited from querying a server of the
 * updater.Realmserv class by calling the del method of the realm.Reader endpoint. */
request dst=updater.Realmserv,
        endpoint=realm.Reader {
            match method=del {
                mic.delete { source : src_sid
                           , target : message.handle.handle
                           , container : ()
                           , driver : dst_sid
                           }
               }
}
```

Mic security model execute rule

```
execute <ExecuteImage | ExecuteLevel>

type ExecuteImage =
  { image  : Sid
  , target : Sid
  , level  : Level | ... | ()
  , levelR : Level | ... | ()
  }

type ExecuteLevel =
  { image  : Sid | ()
  , target : Sid
  , level  : Level | ...
  , levelR : Level | ... | ()
  }
```

This assigns the specified integrity `level` to the `target` subject and defines the minimum integrity level of subjects and resources from which this subject can receive data (`levelR`). The code of the `target` subject is in the `image` executable file.

If the `level` field has the value `()`, the integrity level of the `image` executable file is assigned to the `target` subject. If the `image` field has the value `()`, the `level` field must have a value other than `()`.

If the `levelR` field has the value `()`, the `levelR` integrity level is assumed to be equal to the integrity level of the `target` subject.

To define the integrity `level` and `levelR`, values of the `Level` type are used. For the definition of the `Level` type, see Mic security model create rule.

The rule returns the "granted" result if it assigned the specified integrity `level` to the `target` subject and defined the minimum integrity level of subjects and resources from which this subject can receive data (`levelR`).

The rule returns the "denied" result in the following cases:

- The `level` value exceeds the integrity level of the `image` executable file.
- The `level` value is incomparable to the integrity level of the `image` executable file.
- The value of `levelR` exceeds the value of `level`.
- The `level` and `levelR` values are incomparable.
- An integrity level was not assigned to the `image` executable file.
- The `image` or `target` value is outside of the permissible range.

Example:

```
/* A process of the updater.Manager class will be allowed to start
 * if, at startup initiation, this process will be assigned
 * the integrity level LOW, and the minimum
 * integrity level will be defined for the processes and resources from which this
 * process can received data (LOW). Otherwise the startup of a process
 * of the updater.Manager class will be denied. */
execute src=Einit, dst=updater.Manager, method=main {
    mic.execute { target : dst_sid
                , image : ()
                , level : "LOW"
                , levelR : "LOW"
                }
}
```

Mic security model upgrade rule

```
upgrade { source    : <Sid>
        , target    : <Sid>
        , container : <Sid | ()>
        , driver    : <Sid>
        , level     : <Level | ...>
        }
```

This elevates the previously assigned integrity level of the `target` resource to the specified `level` in the following situation:

- The `source` process initiates elevation of the integrity level of the `target` resource.
- The `target` resource is managed by the `driver` subject, which is the resource provider or the KasperskyOS kernel.
- The `container` resource is a container for the `target` resource (for example, a directory is a container for files and/or other directories).

If the `container` field has the value `()`, the `target` resource is considered to be the root resource, which means that it has no container.

To define the integrity `level`, values of the `Level` type are used. For the definition of the `Level` type, see Mic security model create rule.

The rule returns the "granted" result if it elevated the previously assigned integrity level of the `target` resource to the `level` value.

The rule returns the "denied" result in the following cases:

- The `level` value does not exceed the integrity level of the `target` resource.
- The `level` value exceeds the integrity level of the `source` process, `driver` subject or `container` resource.
- The integrity level of the `target` resource exceeds the integrity level of the `source` process.
- An integrity level was not assigned to the `source` process, `driver` subject, or `container` resource.
- The value of `source`, `target`, `container` or `driver` is outside of the permissible range.

Mic security model call rule

```
call {source : <Sid>, target : <Sid>}
```

This verifies the permissibility of data streams from the `target` subject to the `source` subject.

It returns the "allowed" result in the following cases:

- The integrity level of the `source` subject does not exceed the integrity level of the `target` subject.
- The integrity level of the `source` subject exceeds the integrity level of the `target` subject, but the minimum integrity level of subjects and resources from which the `source` subject can receive data does not exceed the integrity level of the `target` subject.
- The integrity level of the `source` subject is incomparable to the integrity level of the `target` subject, but the minimum integrity level of subjects and resources from which the `source` subject can receive data does not exceed the integrity level of the `target` subject.

It returns the "denied" result in the following cases:

- The integrity level of the `source` subject exceeds the integrity level of the `target` subject, and the minimum integrity level of subjects and resources from which the `source` subject can receive data exceeds the integrity level of the `target` subject.
- The integrity level of the `source` subject exceeds the integrity level of the `target` subject, and the minimum integrity level of subjects and resources from which the `source` subject can read data is incomparable to the integrity level of the `target` subject.
- The integrity level of the `source` subject is incomparable to the integrity level of the `target` subject, and the minimum integrity level of subjects and resources from which the `source` subject can receive data exceeds the integrity level of the `target` subject.
- The integrity level of the `source` subject is incomparable to the integrity level of the `target` subject, and the minimum integrity level of subjects and resources from which the `source` subject can receive data is incomparable to the integrity level of the `target` subject.

- An integrity level was not assigned to the `source` subject or to the `target` subject.
- The `source` or `target` value is outside of the permissible range.

Example:

```
/* Any client in the solution is allowed to query
 * any server (kernel) if data streams from
 * the server (kernel) to the client are permitted by the
 * Mic security model. Otherwise any client in the solution
 * is prohibited from querying any server (kernel). */
request {
    mic.call { source : src_sid
             , target : dst_sid
             }
}
```

Mic security model invoke rule

```
invoke {source : <Sid>, target : <Sid>}
```

This verifies the permissibility of data streams from the `source` subject to the `target` subject.

It returns the "granted" result if the integrity level of the `target` subject does not exceed the integrity level of the `source` subject.

It returns the "denied" result in the following cases:

- The integrity level of the `target` subject exceeds the integrity level of the `source` subject.
- The integrity level of the `target` subject is incomparable to the integrity level of the `source` subject.
- An integrity level was not assigned to the `source` subject or to the `target` subject.
- The `source` or `target` value is outside of the permissible range.

Mic security model read rule

```
read {source : <Sid>, target : <Sid>}
```

This verifies that the `source` resource consumer is allowed to read data from the `target` resource.

It returns the "allowed" result in the following cases:

- The integrity level of the `source` resource consumer does not exceed the integrity level of the `target` resource.
- The integrity level of the `source` resource consumer exceeds the integrity level of the `target` resource, but the minimum integrity level of subjects and resources from which the `source` resource consumer can receive data does not exceed the integrity level of the `target` resource.

- The integrity level of the `source` resource consumer is incomparable to the integrity level of the `target` resource, but the minimum integrity level of subjects and resources from which the `source` resource consumer can receive data does not exceed the integrity level of the `target` resource.

It returns the "denied" result in the following cases:

- The integrity level of the `source` resource consumer exceeds the integrity level of the `target` resource, and the minimum integrity level of subjects and resources from which the `source` resource consumer can receive data exceeds the integrity level of the `target` resource.

- The integrity level of the `source` resource consumer exceeds the integrity level of the `target` resource, and the minimum integrity level of subjects and resources from which the `source` resource consumer can receive data is incomparable to the integrity level of the `target` resource.

- The integrity level of the `source` resource consumer is incomparable to the integrity level of the `target` resource, and the minimum integrity level of subjects and resources from which the `source` resource consumer can receive data exceeds the integrity level of the `target` resource.

- The integrity level of the `source` resource consumer is incomparable to the integrity level of the `target` resource, and the minimum integrity level of subjects and resources from which the `source` resource consumer can receive data is incomparable to the integrity level of the `target` resource.

- An integrity level was not assigned to the `source` resource consumer or to the `target` resource.

- The `source` or `target` value is outside of the permissible range.

Example:

```
/* Any client in the solution is allowed to query a server of
 * the updater.Realmserv class by calling the read method of the
 * realm.Reader service if the Mic security model permits
 * this client to read data from the resource needed by
 * this client. Otherwise any client in the solution is prohibited from
 * querying a server of the updater.Realmserv class by calling
 * the read method of the realm.Reader endpoint. */
request dst=updater.Realmserv,
        endpoint=realm.Reader {
            match method=read {
                mic.read { source : src_sid
                         , target : message.handle.handle
                         }
               }
}
```

Mic security model write rule

```
write {source : <Sid>, target : <Sid>}
```

This verifies that the `source` resource consumer is allowed to write data to the `target` resource.

It returns the "granted" result if the integrity level of the `target` resource does not exceed the integrity level of the `source` resource consumer.

It returns the "denied" result in the following cases:

- The integrity level of the `target` resource exceeds the integrity level of the `source` resource consumer.

- The integrity level of the `target` resource is incomparable to the integrity level of the `source` resource consumer.
- An integrity level was not assigned to the `source` resource consumer or to the `target` resource.
- The `source` or `target` value is outside of the permissible range.

Mic security model query_level expression

```
query_level {source : <Sid>}
```

It is intended to be used as an expression that verifies fulfillment of the conditions in the `choice` construct (for details on the `choice` construct, see Binding methods of security models to security events). It checks the integrity level of the `source` resource or subject. Depending on the results of this check, various options for security event handling can be performed.

It runs incorrectly in the following cases:

- An integrity level was not assigned to the subject or `source` resource.
- The `source` value is outside of the permissible range.

When the expression runs incorrectly, the Kaspersky Security Module returns the "denied" decision.

Using templates when creating a KasperskyOS-based solution security policy description

When creating a policy description, you can use the Ginger template engine that is integrated into the `nk-psl-gen-c` compiler. To manage the template engine, you must use the parameters of the `nk-psl-gen-c` compiler or parameters of the CMake commands `build_kos_qemu_image()` and `build_kos_hw_image()`.

A Ginger PSL template is a PSL file that contains syntactic constructs of the Ginger template engine in addition to declarations in PSL. Files such as `security.psl.in`, `security.psl`, and `core.psl` are considered to be Ginger PSL templates if they contain syntactic constructs of the Ginger template engine. (A `security.psl.in` file simultaneously serves as a PSL template in a different context because this file may use macros and variables that are not linked to the Ginger template engine.)

A description of the syntactic constructs of the Ginger template engine is provided in the official documentation for this template engine. For example, you can use macros.

Example of defining a macro:

```
{% macro VMMSafe(src) %}
        match method=Allocate {
            match src={{ src }} { grant () }
        }
{% endmacro %}
```

Example macro call

```
{{ VMMSafe(["Einit"]) }}
```

This call will create the following construct:

```
match method=Allocate {
    match src=Einit { grant () }
}
```

Example definition of a macro with loops:

```
{% macro VMMSafe(srcs) %}
{% for method in ["Allocate", "Commit", "Decommit"] %}
        match method={{ method }} {
{% for src in srcs %}
            match src={{ src }} { grant () }
{% endfor %}
        }
{% endfor %}
{% endmacro %}
```

Example construct with a macro call:

```
match endpoint=vmm.VMM {
{{ VMMSafe(["Einit", "client.Client", "consumer.Consumer",
    "kl.bc.BlobContainer", "kl.core.NameServer",
    "kl.VfsEntity", "scm.Producer", "execution_manager.ExecMgrEntity",
    "scm.SecurityConfigMgrEntity", "kl.drivers.ATA", "kl.drivers.PCIE"]) }}
}
```

When the macro is expanded in this construct, the following construct will be created:

```
match endpoint=vmm.VMM {
        match method=Allocate {
match src = Einit                            { grant () }
            match src=client.Client { grant () }
            match src=consumer.Consumer { grant () }
            match src=kl.bc.BlobContainer { grant () }
            match src=kl.core.NameServer { grant () }
            match src=kl.VfsEntity { grant () }
            match src=scm.Producer { grant () }
            match src=execution_manager.ExecMgrEntity { grant () }
            match src=scm.SecurityConfigMgrEntity { grant () }
            match src=kl.drivers.ATA { grant () }
            match src=kl.drivers.PCIE { grant () }
        }
        match method=Commit {
match src = Einit                            { grant () }
            match src=client.Client { grant () }
            match src=consumer.Consumer { grant () }
            match src=kl.bc.BlobContainer { grant () }
            match src=kl.core.NameServer { grant () }
            match src=kl.VfsEntity { grant () }
            match src=scm.Producer { grant () }
            match src=execution_manager.ExecMgrEntity { grant () }
            match src=scm.SecurityConfigMgrEntity { grant () }
            match src=kl.drivers.ATA { grant () }
            match src=kl.drivers.PCIE { grant () }
        }
        match method=Decommit {
match src = Einit                            { grant () }
            match src=client.Client { grant () }
            match src=consumer.Consumer { grant () }
            match src=kl.bc.BlobContainer { grant () }
            match src=kl.core.NameServer { grant () }
            match src=kl.VfsEntity { grant () }
            match src=scm.Producer { grant () }
            match src=execution_manager.ExecMgrEntity { grant () }
            match src=scm.SecurityConfigMgrEntity { grant () }
            match src=kl.drivers.ATA { grant () }
            match src=kl.drivers.PCIE { grant () }
```

```
        }
    }
```

Comments can be used in Ginger PSL templates:

```
{# Comment that will be deleted when the template is expanded. #}
```

A file inclusion construct can also be used in Ginger PSL templates.

Example of a file inclusion construct:

```
{% include "defs.ginger" %}
```

The `defs.ginger` file may contain definitions of macros used in a Ginger PSL template, for example. This file must be located in the same directory as the Ginger PSL template that contains this construct.

Methods of KasperskyOS core endpoints

From the perspective of the Kaspersky Security Module, the KasperskyOS kernel is a container of components that provide endpoints. The list of kernel components is provided in the `Core.edl` file located in the `sysroot-*-kos/include/kl/core` directory of the KasperskyOS SDK. This directory also contains the CDL and IDL files for the formal specification of the kernel.

Methods of core endpoints can be divided into secure methods and potentially dangerous methods. Potentially dangerous methods could be used by a cybercriminal in a compromised solution component to cause a denial of service, set up covert data transfer, or hijack an I/O device. Secure methods cannot be used for these purposes.

Access to methods of core endpoints must be restricted as much as possible by the solution security policy (according to the Least Privilege principle). For that, the following requirements must be fulfilled:

1. Access to a secure method must be granted only to the solution components that require this method.

2. Access to a potentially dangerous method must be granted only to the trusted solution components that require this method.

3. Access to a potentially dangerous method must be granted to untrusted solution components that require this method only if the verifiable access conditions limit the possibilities of malicious use of this method, or if the impact from malicious use of this method is acceptable from a security perspective.

   For example, an untrusted component may be allowed to use a limited set of I/O ports that do not allow this component to take control of I/O devices. In another example, covert data transfer between untrusted components may be acceptable from a security perspective.

Virtual memory endpoint

This endpoint is intended for managing virtual memory.

Information about methods of the endpoint is provided in the table below.

Methods of the vmm.VMM endpoint (kl.core.VMM interface)

Method	Method purpose and parameters	Potential danger of the method

`Allocate`	Purpose Allocates (reserves and optionally commits) a virtual memory region. Parameters • [in] `addr` – preferred base address of the virtual memory region, or `0` for the base address to be selected automatically. • [in] `size` – size of the virtual memory region in bytes. • [in] `flags` – flags defining the parameters of the virtual memory region. • [out] `va` – base address of the allocated virtual memory region. • [out] `rc` – return code.	Allows the following: • Exhaust the kernel memory by creating a multitude of objects within it. • Exhaust the RAM.
`Commit`	Purpose Commits a virtual memory region that was reserved by the `Allocate` method. Parameters • [in] `va` – base address of the virtual memory region. • [in] `size` – size of the virtual memory region in bytes. • [out] `rc` – return code.	Exhausts RAM.
`Decommit`	Purpose Decommits a virtual memory region. Parameters • [in] `va` – base address of the virtual memory region. • [in] `size` – size of the virtual memory region in bytes. • [out] `rc` – return code.	N/A
`Protect`	Purpose Modifies the access rights to the virtual memory region. Parameters • [in] `va` – base address of the virtual memory region. • [in] `size` – size of the virtual memory region in bytes. • [in] `flags` – flags defining the access rights to the virtual memory region. • [out] `rc` – return code.	N/A
`Free`	Purpose	N/A

	Frees up the virtual memory region. Parameters • [in] `va` – base address of the virtual memory region. • [in] `size` – size of the virtual memory region in bytes. • [out] `rc` – return code.	
`Query`	Purpose Gets information about a virtual memory page. Parameters • [in] `va` – address included in the virtual memory page. • [out] `info` – sequence containing information about a virtual memory page. • [out] `rc` – return code.	N/A
`Reset`	Purpose Decommits a virtual memory region. Parameters • [in] `va` – base address of the virtual memory region. • [in] `size` – size of the virtual memory region in bytes. • [out] `rc` – return code.	N/A
`MdlCreate`	Purpose Creates an MDL buffer. Parameters • [in] `size` – size of the MDL buffer in bytes. • [in] `prot` – flags defining the access rights to the MDL buffer. • [out] `handle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the MDL buffer. • [out] `rc` – return code.	Allows the following: • Exhaust the kernel memory by creating a multitude of objects within it. • Exhaust the RAM.
`MdlCreateFromVm`	Purpose Creates an MDL buffer from physical memory that is mapped to the defined virtual memory region and maps the created MDL buffer to this region. Parameters • [in] `va` – base address of the virtual memory region. • [in] `size` – size of the virtual memory region in bytes.	Allows the following: • Exhaust the kernel memory by creating a multitude of objects within it. • Exhaust the RAM.

	• [in] `flags` – flags defining the access rights to the MDL buffer. • [out] `handle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the MDL buffer. • [out] `rc` – return code.	
`MdlGetSize`	Purpose Gets the size of the MDL buffer. Parameters • [in] `handle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the MDL buffer. • [out] `size` – size of the MDL buffer in bytes. • [out] `rc` – return code.	N/A
`MdlMap`	Purpose Reserves a virtual memory region and maps the MDL buffer to it. Parameters • [in] `handle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the MDL buffer. • [in] `offset` – offset (in bytes) in the MDL buffer where mapping should start. • [in] `length` – size (in bytes) of the part of the MDL buffer that needs to be mapped. • [in] `hint` – preferred base address of the virtual memory region, or `0` for the base address to be selected automatically. • [in] `prot` – flags defining the parameters of the virtual memory region. • [out] `address` – base address of the virtual memory region. • [out] `rc` – return code.	Allows the following: • Create shared memory for interprocess communication concealed from the security module if multiple processes own the handles of one MDL buffer (the handle permissions masks must allow mapping of the MDL buffer). • Exhaust the kernel memory by creating a multitude of objects within it.
`MdlClone`	Purpose Creates an MDL buffer based on an existing one. The MDL buffer is created from the same regions of physical memory as the original buffer. Parameters • [in] `originHandle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the original MDL buffer. • [in] `offset` – offset (in bytes) in the original MDL buffer where duplication should start.	Allows the kernel memory to be used up by creating a multitude of objects within it.

	• [in] `length` – size (in bytes) of the part of the original MDL buffer that needs to be duplicated. • [out] `cloneHandle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the created MDL buffer. • [out] `rc` – return code.	

I/O endpoint

This endpoint is intended for working with I/O ports, MMIO, DMA, and interrupts.

Information about methods of the endpoint is provided in the table below.

Methods of the io.IO endpoint (kl.core.IO interface)

Method	Method purpose and parameters	Potential danger of the method
`RegisterPort`	Purpose Registers a sequence of I/O ports. Parameters • [in] `base` – address of the first I/O port in the sequence. • [in] `size` – number of I/O ports in the sequence. • [out] `resource` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the sequence of I/O ports. • [out] `rc` – return code.	Allows the following: • Hijack I/O ports (it is recommended to monitor the address of the first I/O port and the number of I/O ports in the sequence). • Exhaust the kernel memory by creating a multitude of objects within it.
`RegisterMmio`	Purpose Registers an MMIO memory region. Parameters • [in] `base` – base address of the MMIO memory region. • [in] `size` – size of the MMIO memory region in bytes. • [out] `resource` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the MMIO memory region. • [out] `rc` – return code.	Allows the kernel memory to be used up by creating a multitude of objects within it.
`RegisterDma`	Purpose Creates a DMA buffer. Parameters • [in] `size` – size of the DMA buffer in bytes. • [in] `flags` – flags defining the DMA buffer parameters. • [in] `order` – parameter defining the minimum number of memory pages (*2^order*) in a block.	Allows the following: • Exhaust the kernel memory by creating a multitude of objects within it. • Exhaust the RAM.

	• [out] `resource` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the DMA buffer. • [out] `rc` – return code.	
`RegisterIrq`	Purpose Registers an interrupt. Parameters • [in] `irq` – interrupt number. • [out] `resource` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the interrupt. • [out] `rc` – return code.	Allows the kernel memory to be used up by creating a multitude of objects within it.
`MapMem`	Purpose Reserves the virtual memory region and maps the MMIO memory region to it. Parameters • [in] `resource` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the MMIO memory region. • [in] `prot` – flags defining the access rights to the virtual memory region. • [in] `attr` – flags defining the parameters of the virtual memory region (for example, use of caching). • [out] `address` – base address of the virtual memory region. • [out] `mapping` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle is used to free the virtual memory region. • [out] `rc` – return code.	Allows the following: • Take control of a device when mapping an MMIO memory region to a virtual memory region (it is recommended to monitor the base address and size of the MMIO memory region when the `RegisterMmio` method is called). • Create shared memory for interprocess communication concealed from the security module if multiple processes own the handles of one MMIO memory region (the handle permissions masks must allow mapping of the MMIO memory region). • Exhaust the kernel memory by creating a multitude of objects within it.
`PermitPort`	Purpose Opens access to I/O ports. Parameters • [in] `resource` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the sequence of I/O ports. • [out] `access` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle is used to close access to I/O ports. • [out] `rc` – return code.	Allows the following: • Take control of a device (it is recommended to monitor the address of the first I/O port and the number of I/O ports in the sequence when the `RegisterPort` method is called). • Exhaust the kernel memory by creating a multitude of objects within it.
`AttachIrq`	Purpose Attaches the calling thread to an interrupt. Parameters • [in] `resource` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the interrupt. • [in] `flags` – flags defining the interrupt parameters.	Allows the following: • Take CPU time from all other threads, including from other processes (the thread that attached to the interrupt will become a real-time thread). • Make it impossible to terminate a process from another process (the process whose thread was attached to the interrupt cannot be terminated from another process).

	• [out] `delivery` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle is the client IPC handle that is used by the interrupt handler. • [out] `rc` – return code.	• Stop the operating system (if an unhandled exception occurs in the thread handling an interrupt, the operating system stops). • Lock, delay, or incorrectly handle an interrupt (it is recommended to monitor the interrupt number when the `RegisterIrq` method is called). • Exhaust the kernel memory by creating a multitude of objects within it.
`DetachIrq`	Purpose Sends a request to a thread. When this request is fulfilled, the thread must detach from the interrupt. Parameters • [in] `resource` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the interrupt. • [out] `rc` – return code.	Stops interrupt handling in another process.
`EnableIrq`	Purpose Allows (unmasks) an interrupt. Parameters • [in] `resource` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the interrupt. • [out] `rc` – return code.	Allows an interrupt at the system level.
`DisableIrq`	Purpose Denies (masks) an interrupt. Parameters • [in] `resource` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the interrupt. • [out] `rc` – return code.	Denies an interrupt at the system level.
`ModifyDma`	Purpose Modifies the DMA buffer cache settings. Parameters • [in] `resource` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the DMA buffer. • [in] `flags` – flags defining the DMA buffer caching parameters. • [out] `rc` – return code.	N/A
`MapDma`	Purpose Reserves a virtual memory region and maps the DMA buffer to it. Parameters • [in] `resource` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the DMA buffer.	Allows the following: • Create shared memory for interprocess communication concealed from the security module if multiple processes own the handles of one DMA buffer (the handle permissions masks must allow mapping of the DMA buffer).

	• [in] `offset` – offset (in bytes) in the DMA buffer where mapping should start. • [in] `length` – size (in bytes) of the part of the DMA buffer that needs to be mapped. • [in] `hint` – preferred base address of the virtual memory region, or `0` for the base address to be selected automatically. • [in] `prot` – flags defining the access rights to the virtual memory region. • [out] `address` – base address of the virtual memory region. • [out] `mapping` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle is used to free the virtual memory region. • [out] `rc` – return code.	• Exhaust the kernel memory by creating a multitude of objects within it.
`DmaGetInfo`	Purpose Gets information about a DMA buffer. Parameters • [in] `resource` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the DMA buffer. • [out] `flags` – flags indicating the DMA parameters. • [out] `order` – parameter indicating the minimum number of memory pages (2^order) in a block. • [out] `size` – size of the DMA buffer in bytes. • [out] `count` – number of blocks. • [out] `frames` – sequence containing the addresses and sizes of blocks. • [out] `rc` – return code.	N/A
`DmaGetPhysInfo`	Purpose Gets the physical addresses and sizes of DMA buffer blocks. Parameters • [in] `handle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the DMA buffer. • [out] `count` – number of blocks. • [out] `frames` – sequence containing the physical addresses and sizes of blocks. • [out] `rc` – return code.	N/A
`BeginDma`	Purpose Opens access to a DMA buffer for a device. Parameters • [in] `resource` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the DMA buffer. • [in] `domain` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The	Allows the kernel memory to be used up by creating a multitude of objects within it.

	handle identifies the IOMMU domain. • [out] `iomapping` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the kernel object that contains the addresses and sizes of blocks required by the device to use the DMA buffer. The memory addresses used by the device can be physical addresses or virtual addresses depending on whether the IOMMU is enabled. • [out] `rc` – return code.	

Threads endpoint

This endpoint is intended for managing threads.

Information about methods of the endpoint is provided in the table below.

Methods of the thread.Thread endpoint (kl.core.Thread interface)

Method	Method purpose and parameters	Potential danger of the method
`Create`	Purpose Creates a thread. Parameters • [out] `thread` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the thread. • [out] `tid` – thread ID (TID). • [in] `priority` – thread priority. • [in] `stackSize` – thread stack limit (in bytes), or `0` to use the default size that was defined when the process was created. • [in] `routine` – pointer to the function that is called when the thread starts. • [in] `context` – pointer to the function executed by the thread. • [in] `context2` – pointer to the parameters passed to the function defined via the `context` parameter. • [in] `flags` – flags defining the parameters for creating the thread. • [out] `rc` – return code.	Allows the following: • Create a real-time thread that takes up all the CPU time from other threads, including from other processes (it is recommended to monitor thread creation parameters). • Create a multitude of threads (including with high priority) to reduce the CPU time available to the threads of other processes (it is recommended to monitor thread priority). • Exhaust the RAM. • Exhaust the kernel memory by creating a multitude of objects within it.
`OpenCurrent`	Purpose Creates the handle of the calling thread. Parameters • [out] `thread` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the thread. • [out] `rc` – return code.	N/A

`Suspend`	Purpose Locks the calling thread. Parameters • [out] `rc` – return code.	Locks a thread that has captured a synchronization object that was created in shared memory and is anticipated by a thread of another process. As a result, the thread of the other process may be locked indefinitely.
`Resume`	Purpose Resumes execution of a locked thread. Parameters • [in] `thread` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the thread. • [out] `rc` – return code.	N/A
`Terminate`	Purpose Terminates a thread. Parameters • [in] `thread` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the thread. • [in] `code` – thread exit code. • [out] `tid` – thread ID (TID). • [out] `rc` – return code.	N/A
`Exit`	Purpose Terminates the calling thread. Parameters • [in] `code` – thread exit code. • [out] `rc` – return code.	N/A
`Wait`	Purpose Locks the calling thread until the defined thread is terminated. Parameters • [in] `thread` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the thread. • [in] `msec` – thread termination timeout (in milliseconds). • [out] `code` – thread exit code. • [out] `rc` – return code.	N/A
`SetPriority`	Purpose	Allows the priority of a thread to be elevated to reduce the CPU time available to all other threads, including from other

	Defines the priority of a thread. Parameters • [in] `thread` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the thread. • [in] `priority` – thread priority. • [out] `rc` – return code.	processes. It is recommended to monitor thread priority.
`SetTls`	Purpose Defines the base address of the thread local storage (TLS) for the calling thread. Parameters • [in] `va` – pointer to the local memory of the thread. • [out] `rc` – return code.	N/A
`Sleep`	Purpose Locks the calling thread for the specified duration. Parameters • [in] `mdelay` – thread lockout duration (in milliseconds). • [out] `rc` – return code.	N/A
`GetInfo`	Purpose Gets information about a thread. Parameters • [in] `thread` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the thread. • [out] `info` – structure containing the base address of the thread stack and its size (in bytes), and the thread identifier (TID). • [out] `rc` – return code.	N/A
`DetachIrq`	Purpose Detaches the calling thread from the interrupt handled in its context. Parameters • [out] `rc` – return code.	N/A
`GetAffinity`	Purpose Gets a thread affinity mask. Parameters • [in] `thread` – value whose binary representation consists of multiple fields, including a handle field	N/A

	and a handle permissions mask field. The handle identifies the thread. • [out] `mask` – thread affinity mask. • [out] `rc` – return code.	
`SetAffinity`	Purpose Defines a thread affinity mask. Parameters • [in] `thread` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the thread. • [in] `mask` – thread affinity mask. • [out] `rc` – return code.	N/A
`SetSchedPolicy`	Purpose Defines the scheduler class and priority of the thread. Parameters • [in] `thread` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the thread. • [in] `policy` – thread scheduler class. • [in] `priority` – thread priority. • [in] `param` – union containing parameters of a thread scheduler class. • [out] `rc` – return code.	Allows the following: • Convert a thread into a real-time thread that takes up all the CPU time from all other threads, including from other processes (it is recommended to monitor the thread scheduler class). • Elevate the priority of a thread to reduce the CPU time available to all other threads, including from other processes (it is recommended to monitor thread priority).
`GetSchedPolicy`	Purpose Gets information about the scheduler class and priority of a thread. Parameters • [in] `thread` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the thread. • [out] `policy` – thread scheduler class. • [out] `priority` – thread priority. • [out] `param` – union containing parameters of a thread scheduler class. • [out] `rc` – return code.	N/A

Handles endpoint

This endpoint is intended for performing operations with handles.

Information about methods of the endpoint is provided in the table below.

Methods of the handle.Handle endpoint (kl.core.Handle interface)

Method	Method purpose and parameters	Potential danger of the method
`Copy`	Purpose Duplicates a handle. As a result of duplication, the calling process receives the handle descendant. Parameters • [in] `inHandle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle field contains the original handle. • [in] `newRightsMask` – permissions mask of the handle descendant. • [in] `copyBadge` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the resource transfer context object. • [out] `outHandle` – value whose binary representation consists of multiple fields, including a field for the handle descendant and a field for the permissions mask of the handle descendant. • [out] `rc` – return code.	Allows the kernel memory to be used up by creating a multitude of objects within it.
`CreateUserObject`	Purpose Creates a handle. Parameters • [in] `type` – handle type. • [in] `rights` – handle permissions mask. • [in] `context` – pointer to the data that should be associated with the handle. • [in] `ipcChannel` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle is the server IPC handle. • [in] `riid` – Runtime Implementation Identifier (RIID). • [out] `handle` – value whose binary representation consists of multiple fields, including a field for the created handle and a field for the permissions mask of the created handle. • [out] `rc` – return code.	Allows the kernel memory to be used up by creating a multitude of objects within it.
`Close`	Purpose Closes a handle. Parameters • [in] `handle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. • [out] `rc` – return code.	N/A
`Connect`	Purpose Creates and connects the client, server, and listener IPC handles. Parameters • [in] `server` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the server process.	Allows the kernel memory to be used up by creating a multitude of objects within it.

	• [in] `srListener` – listener handle from the handle space of the server process, or the value `0xFFFFFFFF` to create it. • [in] `createSrEndpoint` – value that defines whether or not to create a server IPC handle in the handle space of the server process (`0` means no, and any other number means yes). • [in] `client` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the client process. • [out] `outSrListener` – listener handle from the handle space of the server process. • [out] `outSrEndpoint` – server IPC handle from the handle space of the server process. • [out] `outClEndpoint` – client IPC handle from the handle space of the client process. • [out] `rc` – return code.	
`SecurityConnect`	Purpose Creates a client IPC handle for querying the Kaspersky Security Module through the security interface. Parameters • [out] `client` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. • [out] `rc` – return code.	Allows a multitude of possible kernel process handle values to be used up.
`GetSidByHandle`	Purpose Receives a security ID (SID) based on a handle. Parameters • [in] `handle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. • [out] `sid` – security ID (SID). • [out] `rc` – return code.	N/A
`Revoke`	Purpose Closes a handle and revokes its descendants. Parameters • [in] `handle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. • [out] `rc` – return code.	N/A
`RevokeSubtree`	Purpose Revokes the handles that make up the inheritance subtree of the specified handle. Parameters • [in] `handle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handles forming the inheritance subtree of this handle are revoked. • [in] `badge` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the resource transfer context object that defines the inheritance subtree of the handles to revoke. The root node of this subtree is the handle that was generated by the transfer or duplication of the handle that is defined through the `handle` parameter and is associated with the resource transfer context object. • [out] `rc` – return code.	N/A

`CreateBadge`	Purpose Creates a resource transfer context object and configures a notification mechanism for monitoring the life cycle of this object. Parameters • [in] `notify` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the notification receiver. • [in] `notifyContext` – ID of the "resource–event mask" entry in the notification receiver. • [in] `badgeContext` – pointer to the data that should be associated with the handle transfer. • [out] `badge` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the resource transfer context object. • [out] `rc` – return code.	Allows the kernel memory to be used up by creating a multitude of objects within it.

Processes endpoint

This endpoint is intended for managing processes.

Information about methods of the endpoint is provided in the table below.

Methods of the task.Task endpoint (kl.core.Task interface)

Method	Method purpose and parameters	Potential danger of the method
`Create`	Purpose Creates a process. Parameters • [in] `name` – process name. • [in] `eiid` – process class name. • [in] `path` – name of the executable file in ROMFS. • [in] `stackSize` – thread stack limit (in bytes) used by default when creating process threads. • [in] `priority` – priority of the initial thread. • [in] `flags` – flags defining the parameters for creating the process. • [out] `child` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the created process. • [out] `rc` – return code.	Allows the following: • Create a process that will be privileged from the perspective of the solution security policy (indicating the name of the process class with privileges). • Reserve a process name so that another process with this name cannot be created. • Create a process that will cause the operating system to stop if an unhandled exception occurs. • Load code from an executable file into process memory for subsequent execution of that code. • Exhaust RAM by creating a multitude of processes. • Exhaust the kernel memory by creating a multitude of objects within it.
`LoadSeg`	Purpose Loads an ELF image segment into process memory from the MDL buffer. Parameters • [in] `task` – value whose binary representation consists of multiple fields, including a handle field	Allows code to be loaded into process memory for subsequent execution of that code.

	and a handle permissions mask field. The handle identifies the process. • [in] `mdl` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the MDL buffer containing the ELF image segment. • [in] `segAttr` – structure containing the parameters for loading the ELF image segment. • [out] `rc` – return code. • [out] `retaddr` – base address of the virtual memory region of the process where the ELF image segment is loaded.	
`VmReserve`	Purpose Reserves the virtual memory region in a process that was created as an empty process. Parameters • [in] `task` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the process. • [in] `addr` – preferred base address of the virtual memory region, or `0` for the address to be selected automatically. • [in] `size` – size of the virtual memory region in bytes. • [in] `flags` – flags defining the parameters of the virtual memory region. • [out] `outAddr` – base address of the reserved virtual memory region. • [out] `rc` – return code.	Allows the following: • Exhaust the kernel memory by creating a multitude of objects within it. • Reserve virtual memory regions in another process that was created as an empty process and has not yet been started (if its handle is available). The handle permissions mask must allow reservation of virtual memory.
`VmFree`	Purpose Frees the virtual memory region that was reserved by calling the `VmReserve` method in a process that was created as an empty process. Parameters • [in] `task` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the process. • [in] `addr` – base address of the virtual memory region. • [in] `size` – size of the virtual memory region in bytes. • [out] `rc` – return code.	Frees virtual memory regions in another process that was created as an empty process and has not yet been started (if its handle is available). (The handle permissions mask must allow freeing of virtual memory.)
`SetEntry`	Purpose Defines the program entry point and the ELF image load offset. Parameters	Creates conditions for executing code loaded into process memory.

	• [in] `task` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the process. • [in] `state` – structure containing the address of the program entry point and the ELF image load offset (in bytes). • [out] `rc` – return code.	
`LoadElfSyms`	Purpose Loads the symbol table `.symtab` and string table `.strtab` from MDL buffers into the memory of a process that was created as an empty process. Parameters • [in] `task` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the process. • [in] `symMdl` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the MDL buffer containing the symbol table `.symtab`. • [in] `symSegAttr` – structure containing the parameters for loading the symbol table `.symtab`. • [in] `symSize` – size of the symbol table `.symtab` (in bytes). • [in] `strMdl` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the MDL buffer containing the string table `.strtab`. • [in] `strSegAttr` – structure containing the parameters for loading the string table `.strtab`. • [in] `strSize` – size of the string table `.strtab` (in bytes). • [out] `rc` – return code.	N/A
`LoadElfHdr`	Purpose Writes the ELF image header to the PCB of a process that was created as an empty process. Parameters • [in] `task` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the process. • [in] `hdrData` – sequence containing the ELF image header. • [out] `rc` – return code.	N/A
`SetEnv`	Purpose Writes data to the SCP of a child process. Parameters	Allows the kernel memory to be used up by creating a multitude of objects within it.

	• [in] `task` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the child process. • [in] `env` – sequence containing data to be written to the SCP. • [out] `rc` – return code.	
`FreeSelfEnv`	Purpose Deletes the SCP of the calling process. Parameters • [out] `rc` – return code.	N/A
`Resume`	Purpose Starts a process. Parameters • [in] `task` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the process. • [out] `rc` – return code.	Allows the following: • Execute code loaded into process memory. • Start a multitude of previously created processes to reduce the computing resources available to other processes (it is recommended to monitor the priority of the initial thread when the `Create` method is called).
`Exit`	Purpose Terminates the calling process. Parameters • [in] `status` – exit code of the process. • [out] `rc` – return code.	N/A
`Terminate`	Purpose Terminates a process. Parameters • [in] `task` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the process. • [out] `rc` – return code.	Allows another process to be terminated if its handle is available. (The handle permissions mask must allow termination of the process.)
`GetExitInfo`	Purpose Gets information about a terminated process. Parameters • [in] `task` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the terminated process. • [out] `status` – value indicating the reason for process termination. • [out] `info` – union containing information about the terminated process.	N/A

	• [out] `rc` – return code.	
`GetThreadContext`	Purpose Gets the context of a thread that is part of a frozen process. Parameters • [in] `task` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the process that is in a frozen state. • [in] `index` – thread index. It is used to enumerate threads. Enumeration starts with zero. A thread in which an unhandled exception occurred has a zero index. • [out] `context` – structure containing the thread ID (TID) and thread context. • [out] `rc` – return code.	Enables disrupted isolation of a process that is in a frozen state. For example, the thread context may contain the values of variables.
`GetNextVmRegion`	Purpose Gets information about the virtual memory region that belongs to a frozen process. Parameters • [in] `task` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the process that is in a frozen state. • [in] `after` – address that is followed by the virtual memory region. • [out] `next` – base address of the virtual memory region. • [out] `size` – size of the virtual memory region in bytes. • [out] `flags` – flags indicating the parameters of the virtual memory region. • [out] `handle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the MDL buffer mapped to a virtual memory region. • [out] `rc` – return code.	Enables disrupted isolation of a process that is in a frozen state. Process isolation is disrupted due to the opened access to the process memory region.
`TerminateAfterFreezing`	Purpose Terminates a frozen process. Parameters • [in] `task` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the process that is in a frozen state. • [out] `rc` – return code.	Enables termination of a frozen process. This does not allow collection of data about this process for diagnostic purposes.
`GetName`	Purpose	N/A

	Gets the name of a calling process. Parameters • [out] `name` – process name. • [out] `rc` – return code.	
`GetPath`	Purpose Gets the name of the executable file (in ROMFS) that was used to create the calling process. Parameters • [out] `path` – name of the executable file. • [out] `rc` – return code.	N/A
`GetInitialThreadPriority`	Purpose Gets the priority of the initial thread of a process. Parameters • [in] `task` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the process. • [out] `priority` – priority of the initial thread. • [out] `rc` – return code.	N/A
`SetInitialThreadPriority`	Purpose Defines the priority of the initial thread of a process. Parameters • [in] `task` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the process. • [in] `priority` – priority of the initial thread. • [out] `rc` – return code.	Allows the priority of the initial thread of a process to be elevated to reduce the CPU time available to all other threads, including from other processes. It is recommended to monitor the priority of an initial thread.
`GetTasksList`	Purpose Gets information about existing processes. Parameters • [out] `notice` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the notification receiver that is configured to receive notifications regarding the termination of processes. • [out] `strings` – sequence containing the parameters of processes. • [out] `pids` – sequence containing the identifiers of processes (the PID of each process). • [out] `rc` – return code.	Allows the kernel memory to be used up by creating a multitude of objects within it.

`SetInitialThreadSchedPolicy`	Purpose Defines the scheduler class and priority of the initial thread of a process. Parameters • [in] `task` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the process. • [in] `policy` – scheduler class of the initial thread of the process. • [in] `priority` – priority of the initial thread of a process. • [in] `params` – union containing the parameters of the scheduler class of the initial thread of the process. • [out] `rc` – return code.	Allows the following: • Convert the initial thread of a process into a real-time thread that takes up all the CPU time from all other threads, including from other processes (it is recommended to monitor the scheduler class of the initial thread of the process). • Elevate the priority of the initial thread of a process to reduce the CPU time available to all other threads, including from other processes (it is recommended to monitor the priority of the initial thread of the process).
`ReseedAslr`	Purpose Defines the seed value for ASLR support. Parameters • [in] `task` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the process. • [in] `seed` – sequence containing the seed value. • [out] `rc` – return code.	N/A
`GetElfSyms`	Purpose Gets the address and size of the symbol table `.symtab` and string table `.strtab` for the calling process. Parameters • [out] `relocBase` – ELF image load offset (in bytes). • [out] `syms` – address of the symbol table `.symtab`. • [out] `symsCnt` – size (in bytes) of the symbol table `.symtab`. • [out] `strs` – address of the string table `.strtab`. • [out] `strsSize` – size (in bytes) of the string table `.strtab`. • [out] `rc` – return code.	N/A
`TransferHandle`	Purpose Transfers a handle to a process that is not yet running. Parameters • [in] `task` – value whose binary representation consists of multiple fields, including a handle field	Allows the kernel memory to be used up by creating a multitude of objects within it.

	and a handle permissions mask field. The handle identifies the process. • [in] `srcHandle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle field contains the transferred handle. • [in] `srcBadge` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the resource transfer context object. • [in] `dstRights` – permissions mask of the descendant of the transferred handle. • [out] `dstHandle` – value of the descendant of the transferred handle (from the handle space of the process that received the handle). • [out] `rc` – return code.	
`GetPid`	Purpose Gets the process ID (PID). Parameters • [in] `task` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the process. • [out] `pid` – process ID. • [out] `rc` – return code.	N/A

Synchronization endpoint

This endpoint is intended for working with futexes.

Information about methods of the endpoint is provided in the table below.

Methods of the sync.Sync endpoint (kl.core.Sync interface)

Method	Method purpose and parameters	Potential danger of the method
`Wait`	Purpose Locks execution of the calling thread if the futex value is equal to the expected value. Parameters • [in] `ptr` – pointer to the variable containing the futex value. • [in] `val` – expected value of the futex. • [in] `delay` – maximum lockout duration in milliseconds. • [out] `outDelay` – actual lockout duration in milliseconds. • [out] `rc` – return code.	N/A
`Wake`	Purpose Resumes execution of threads that were blocked by `Wait` method calls with the defined futex.	N/A

	Parameters • [in] `ptr` – pointer to the variable containing the futex value. • [in] `nThreads` – maximum number of threads whose execution can be resumed. • [out] `wokenCnt` – actual number of threads whose execution was resumed. • [out] `rc` – return code.	

File system endpoints

These endpoints are intended for working with the ROMFS file system used by the KasperskyOS kernel.

Information about methods of endpoints is provided in the tables below.

Methods of the fs.FS endpoint (kl.core.FS interface)

Method	Method purpose and parameters	Potential danger of the method
`Open`	Purpose Opens a file. Parameters • [in] `name` – name of the file. • [out] `handle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the opened file. • [out] `rc` – return code.	Allows the kernel memory to be used up by creating a multitude of objects within it.
`Close`	Purpose Closes a file. Parameters • [in] `handle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the opened file. • [out] `rc` – return code.	N/A
`Read`	Purpose Reads data from a file. Parameters • [in] `handle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the opened file. • [in] `sectorNumber` – data block number. Enumeration starts with zero. • [out] `read` – size of the read data in bytes. • [out] `data` – sequence containing the read data. • [out] `rc` – return code.	N/A
`GetSize`	Purpose	N/A

	Gets the size of a file. Parameters • [in] `handle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the opened file. • [out] `size` – file size in bytes. • [out] `rc` – return code.	
`GetId`	Purpose Gets the unique ID of a file. Parameters • [in] `handle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the opened file. • [out] `id` – unique ID of the file. • [out] `rc` – return code.	N/A
`Count`	Purpose Gets the number of files in the file system. Parameters • [out] `count` – number of files in the file system. • [out] `rc` – return code.	N/A
`GetInfo`	Purpose Gets the name and unique ID of a file based on the file index. Parameters • [in] `index` – file index. Enumeration starts with zero. • [in] `nameLenMax` – buffer size (in bytes) for saving the file name. • [out] `name` – name of the file. • [out] `id` – unique ID of the file. • [out] `rc` – return code.	N/A
`GetFsSize`	Purpose Gets the size of the file system. Parameters • [out] `fsSize` – size of the file system in bytes. • [out] `rc` – return code.	N/A

Methods of the fs.FSUnsafe endpoint (kl.core.FSUnsafe interface)

Method	Method purpose and parameters	Potential danger of the method
`Change`	Purpose Changes the file system image.	Allows the following: • Use an ROMFS image containing arbitrary programs and data.

	A different ROMFS image loaded into process memory will be used instead of the ROMFS image that was created during the solution build. Parameters • [in] `base` – pointer to the file system image. • [in] `size` – size of the file system image in bytes. • [out] `rc` – return code.	• Gain read-access to some kernel objects.

Time endpoint

This endpoint is intended for setting the system time.

Information about methods of the endpoint is provided in the table below.

Methods of the time.Time endpoint (kl.core.Time interface)

Method	Method purpose and parameters	Potential danger of the method
`SetSystemTime`	Purpose Sets the system time. Parameters • [in] `secs` – time (in seconds) that has elapsed since January 1, 1970. • [in] `nsecs` – additional time (in nanoseconds) added to the time defined through the `secs` parameter. • [out] `rc` – return code.	Allows the system time to be set.
`SetSystemTimeAdj`	Purpose Starts gradual adjustment of the system time. Parameters • [in] `adj` – structure containing the amount of time by which the system time must be adjusted (*sec*10^9+nsec* nanoseconds). • [in] `slew` – rate of system time adjustment (microseconds per second). • [out] `prev` – structure containing the correction time value that remained for system time adjustment to be completed for the previous gradual time adjustment (*sec*10^9+nsec* nanoseconds). • [out] `rc` – return code.	Allows the system time to be changed.
`GetSystemTimeAdj`	Purpose Gets the correction time value remaining for system time adjustment so that gradual adjustment can be fully completed. Parameters • [out] `adj` – structure containing the correction time value remaining for system time adjustment so that the gradual adjustment can be fully completed (*sec*10^9+nsec* nanoseconds). • [out] `rc` – return code.	N/A

Hardware abstraction layer endpoint

This endpoint is intended for receiving the values of hardware platform parameters, reading and setting of peripheral devices privileged registers, clearing the processor cache, providing program diagnostic output, and receiving hardware-generated random numbers.

Information about methods of the endpoint is provided in the table below.

Methods of the hal.HAL endpoint (kl.core.HAL interface)

Method	Method purpose and parameters	Potential danger of the method
`GetEnv`	Purpose Gets the value of a hardware platform parameter. Parameters • [in] `name` – name of the parameter. • [out] `value` – value of the parameter. • [out] `rc` – return code.	Gets values of hardware platform parameters that could contain critical system information.
`GetPrivReg`	Purpose Gets the value of a peripheral device privileged register. Parameters • [in] `reg` – name of the register. • [out] `val` – value of the register. • [out] `rc` – return code.	Sets up a data transfer channel with a process that has access to the `SetPrivReg` or `SetPrivRegRange` method. It is recommended to monitor the name of a register.
`SetPrivReg`	Purpose Sets the value of a peripheral device privileged register. Parameters • [in] `reg` – name of the register. • [in] `val` – value of the register. • [out] `rc` – return code.	Allows the following: • Set the value of a peripheral device privileged register. • Set up a data transfer channel with a process that has access to the `GetPrivReg` or `GetPrivRegRange` method. It is recommended to monitor the name of a register.
`GetPrivRegRange`	Purpose Gets the value of a peripheral device privileged register. Parameters • [in] `regRange` – name of the registers range. • [in] `offset` – register offset in the registers range. • [out] `val` – value of the register. • [out] `rc` – return code.	Sets up a data transfer channel with a process that has access to the `SetPrivReg` or `SetPrivRegRange` method. It is recommended to monitor the name of the registers range and the register offset in this range.

`SetPrivRegRange`	Purpose Sets the value of a peripheral device privileged register. Parameters • [in] `regRange` – name of the registers range. • [in] `offset` – register offset in the registers range. • [in] `val` – value of the register. • [out] `rc` – return code.	Allows the following: • Set the value of a peripheral device privileged register. • Set up a data transfer channel with a process that has access to the `GetPrivReg` or `GetPrivRegRange` method. It is recommended to monitor the name of the registers range and the register offset in this range.
`FlushCache`	Purpose Clears the processor cache. Parameters • [in] `type` – value defining the cache type (data cache, instructions cache, or joint data and instructions cache). • [in] `va` – base address of the virtual memory region. The cache corresponding to this region is cleared. • [in] `size` – size of the virtual memory region in bytes. The cache corresponding to this region is cleared. • [out] `rc` – return code.	Allows the processor cache to be cleared.
`DebugWrite`	Purpose Puts data into the diagnostic output that is written, for example, to a COM port or USB port (version 3.0 or later, with DbC support). Parameters • [in] `data` – sequence containing the data to be put into the diagnostic output. • [out] `rc` – return code.	Populates diagnostic output with fictitious (uninformative) data.
`GetEntropy`	Purpose Gets hardware-generated random numbers. Parameters • [out] `buffer` – sequence containing random byte values. • [in] `size` – number of random byte values. • [out] `rc` – return code.	Creates a load on the hardware-based random number generator with frequent method calls so that other processes are unable to receive random numbers using this generator.

KasperskyOS kernel XHCI DbC driver management endpoint

This endpoint is intended for use by drivers that implement a USB stack (kusb class drivers) and lets you stop and start the XHCI DbC driver of the KasperskyOS kernel.

Information about methods of the endpoint is provided in the table below.

Methods of the xhcidbg.XHCIDBG endpoint (kl.core.XHCIDBG interface)

Method	Method purpose and parameters	Potential danger of the method
`Start`	Purpose Starts the XHCI DbC driver of the KasperskyOS kernel. Parameters • [out] `rc` – return code.	Configures the XHCI controller to send diagnostic (or debug) output through a USB port (version 3.0 or later).
`Stop`	Purpose Stops the XHCI DbC driver of the KasperskyOS kernel. Parameters • [out] `rc` – return code.	Configures the XHCI controller to not send diagnostic (or debug) output through a USB port (version 3.0 or later).

Audit endpoint

This endpoint is intended for reading the KasperskyOS kernel log containing security audit data.

Information about methods of the endpoint is provided in the table below.

Methods of the audit.Audit endpoint (kl.core.Audit interface)

Method	Method purpose and parameters	Potential danger of the method
`Open`	Purpose Opens the kernel log containing security audit data. Parameters • [in] `name` – name of the kernel log containing security audit data (`kss`). • [out] `handle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. This handle identifies the kernel log containing security audit data. • [out] `rc` – return code.	N/A
`Close`	Purpose Closes the kernel log containing security audit data. Parameters • [in] `handle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. This handle identifies the kernel log containing security audit data. • [out] `rc` – return code.	N/A
`Read`	Purpose	Reads messages from the kernel log containing security audit data so that these messages are

	Receives a message from the kernel log containing security audit data. Parameters • [in] `handle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. This handle identifies the kernel log containing security audit data. • [out] `msg` – sequence containing a message. • [out] `outDropMsgs` – number of messages that were lost due to overwrites in the kernel log containing security audit data. • [out] `rc` – return code.	not received by another process.

Profiling endpoint

This endpoint is intended for profiling and collecting code coverage, and for receiving the values of performance counters.

Information about methods of the endpoint is provided in the table below.

Methods of the profiler.Profiler endpoint (kl.core.Profiler interface)

Method	Method purpose and parameters	Potential danger of the method
`GetCoverageData`	Purpose Gets information about kernel code coverage. Parameters • [in] `index` – index for enumerating object files containing instrumental code for receiving coverage data. Enumeration starts with zero. • [out] `buf` – sequence containing information about the code coverage of an object file (in gcda format). • [out] `size` – size (in bytes) of data containing information about the code coverage of an object file. • [out] `name` – name of the `*.gcda` file that was assigned during compilation. • [out] `rc` – return code.	N/A
`FlushGcov`	Purpose Output of data on kernel code coverage in gcda format via I/O interface (for example, COM, USB). Parameters • [out] `rc` – return code.	N/A
`FlushGcovFile`	Purpose Output of data on user code coverage in gcda format via I/O interface (for example, COM, USB). Parameters • [in] `name` – name of the `*.gcda` file that was assigned during compilation.	N/A

	• [in] `buf` – pointer to the buffer containing information about code coverage in gcda format. • [in] `size` – size of data containing code coverage information in bytes. • [out] `rc` – return code.	
`GetCounters`	Purpose Gets the values of performance counters. Parameters • [in] `prefix` – prefix for names of performance counters. • [in] `names` – sequence containing the names of performance counters. • [out] `values` – sequence containing the values of performance counters. • [out] `rc` – return code.	N/A
`ObjectGetStat`	Purpose Gets the values of performance counters associated with a system resource (process or thread). Parameters • [in] `handle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the system resource. • [in] `names` – sequence containing the names of performance counters. • [out] `values` – sequence containing the values of performance counters. • [out] `rc` – return code.	N/A
`SamplingStart`	Purpose Starts sample code profiling. Parameters • [in] `conf` – flags that define the profiling settings. • [in] `cpus` – processor mask indicating the set of processors (processor cores) that execute the profiled code. • [in] `contSize` – size (in bytes) of the container used to store data containing the code execution statistics obtained from profiling. • [in] `interval` – fictitious parameter. • [out] `rc` – return code.	N/A
`SamplingStop`	Purpose Stops sample code profiling. Parameters • [out] `rc` – return code.	N/A
`SamplingRead`	Purpose	Gets the addresses and names of functions of other processes.

	Gets data containing the code execution statistics received from sample profiling. Parameters • [in] `unsafeBuffer` – pointer to the buffer used to save the data on the code execution statistics obtained from profiling. • [in] `size` – size of the buffer whose pointer is defined through the `unsafeBuffer` parameter in bytes. • [out] `realSize` – size of the saved data in bytes. • [in] `timeout` – container filling timeout (in milliseconds). • [out] `rc` – return code.	
`SamplingAddPidToList`	Purpose Adds a process to the set of processes in whose contexts the programs subject to sample profiling are running. Parameters • [in] `pid` – process ID (PID). • [out] `rc` – return code.	N/A
`SamplingClearPidList`	Purpose Clears the set of processes in whose contexts the programs subject to sample profiling are running. Parameters • [out] `rc` – return code.	N/A
`LoadSegInfo`	Purpose Saves information about the loaded segment of the ELF image in the kernel. Parameters • [in] `task` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the process. • [in] `addr` – segment address in process memory. • [in] `size` – segment size (in bytes). • [in] `offset` – offset of the segment in the ELF file (in bytes). • [in] `flags` – flags defining the access rights to the segment. • [in] `buildId` – build ID. • [out] `rc` – return code.	Allows the kernel memory to be used up by creating a multitude of objects within it.
`UnloadSegInfo`	Purpose Deletes information about the loaded ELF image segment that was saved in the kernel using the `LoadSegInfo` method. Parameters • [in] `task` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the process.	N/A

	• [in] `addr` – segment address in process memory. • [in] `size` – segment size (in bytes). • [out] `rc` – return code.	
`KcovAlloc`	Purpose Allocates the resources required for collecting kernel code coverage data when handling system calls executed by the calling process. Parameters • [in] `numThreads` – maximum number of threads whose kernel code coverage data will be collected. • [in] `maxPoints` – maximum number of coverage points for one thread. • [out] `rc` – return code.	Exhausts RAM.
`KcovFree`	Purpose Frees the resources required for collecting kernel code coverage data when handling system calls executed by the calling process. Parameters • [out] `rc` – return code.	N/A
`KcovStart`	Purpose Starts the collection of kernel code coverage data when handling system calls executed by the calling thread. Parameters • [out] `rc` – return code.	N/A
`KcovStop`	Purpose Stops the collection of kernel code coverage data when handling system calls executed by the calling thread. Also gets information about kernel code coverage. Parameters • [in] `points` – pointer to the buffer used to store the IDs of coverage points. • [in] `maxPoints` – maximum number of coverage point IDs that can be stored in the buffer defined via the `points` parameter. • [out] `numPoints` – actual number of coverage point IDs stored in the buffer defined via the `points` parameter. • [out] `rc` – return code.	N/A

I/O memory isolation management endpoint

This endpoint is intended for managing the isolation of physical memory regions used by devices on a PCIe bus for DMA. (Isolation is provided by the IOMMU.)

Information about methods of the endpoint is provided in the table below.

Method	Method purpose and parameters	Potential danger of the method
`Attach`	Purpose Attaches a device on a PCIe bus to the automatically created IOMMU domain associated with the calling process. Parameters • [in] `bdf` – address of the device on the PCIe bus in BDF format. • [out] `rc` – return code.	Attaches a device on a PCIe bus managed by another process to an IOMMU domain associated with the calling process, which leads to failure of the device. It is recommended to monitor the address of a device on a PCIe bus.
`Detach`	Purpose Detaches a device on a PCIe bus from the automatically created IOMMU domain associated with the calling process. Parameters • [in] `bdf` – address of the device on the PCIe bus in BDF format. • [out] `rc` – return code.	N/A
`CreateDomain`	Purpose Creates an IOMMU domain associated with the calling process. Parameters • [out] `domain` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the IOMMU domain. • [out] `rc` – return code.	Allows the kernel memory to be used up by creating a multitude of objects within it.
`AttachToDomain`	Purpose Attaches a device on a PCIe bus to the defined IOMMU domain associated with the calling process. Parameters • [in] `bdf` – address of the device on the PCIe bus in BDF format. • [in] `domain` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the IOMMU domain. • [out] `rc` – return code.	Attaches a device on a PCIe bus managed by another process to an IOMMU domain associated with the calling process, which leads to failure of the device. It is recommended to monitor the address of a device on a PCIe bus.
`DetachFromDomain`	Purpose Detaches a device on a PCIe bus from the defined IOMMU domain associated with the calling process. Parameters • [in] `bdf` – address of the device on the PCIe bus in BDF format. • [in] `domain` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the IOMMU domain.	N/A

	• [out] `rc` – return code.	

Connections endpoint

This endpoint is intended for dynamic creation of IPC channels.

Information about methods of the endpoint is provided in the table below.

Methods of the cm.CM endpoint (kl.core.CM interface)

Method	Method purpose and parameters	Potential danger of the method
`Connect`	Purpose Requests to create an IPC channel with a server for use of the defined endpoint. Parameters • [in] `server` – name of the server. • [in] `service` – qualified name of the endpoint. • [in] `msecs` – request fulfillment timeout, in milliseconds. • [out] `handle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle is the client IPC handle. • [out] `id` – Runtime Implementation Identifier (RIID). • [out] `rc` – return code.	Creates a load on a server by sending a large number of requests to create an IPC channel.
`Listen`	Purpose Receives a client request to create an IPC channel for use of an endpoint. Parameters • [in] `filter` – fictitious parameter. • [in] `msecs` – client request timeout, in milliseconds. • [out] `client` – client name. • [out] `service` – qualified name of the endpoint. • [out] `rc` – return code.	N/A
`Drop`	Purpose Rejects a client request to create an IPC channel for use of the defined endpoint. Parameters • [in] `client` – client name. • [in] `service` – qualified name of the endpoint. • [out] `rc` – return code.	N/A
`Accept`	Purpose Accepts a client request to create an IPC channel for use of the defined endpoint. Parameters	N/A

	• [in] `client` – client name. • [in] `service` – qualified name of the endpoint. • [in] `id` – Runtime Implementation Identifier (RIID). • [in] `listener` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle is the listener handle. • [out] `handle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle is the server IPC handle. • [out] `rc` – return code.	

Power management endpoint

This endpoint is intended for changing the power management mode of a hardware platform (for example, shutting down or restarting the hardware platform), and for enabling and disabling processors (processor cores).

Information about methods of the endpoint is provided in the table below.

Methods of the pm.PM endpoint (kl.core.PM interface)

Method	Method purpose and parameters	Potential danger of the method
`Request`	Purpose Requests to change the power mode of a hardware platform. Parameters • [in] `request` – value defining the necessary power mode of the hardware platform. • [out] `rc` – return code.	Allows the hardware platform power mode to be changed.
`SetCpusOnline`	Purpose Requests to enable and/or disable processors. Parameters • [in] `request` – processor mask indicating the required set of active processors. • [out] `rc` – return code.	Disables and enables processors.
`GetCpusOnline`	Purpose Gets information regarding which processors are in the active state. Parameters • [out] `online` – processor mask indicating the actual set of active processors. • [out] `rc` – return code.	N/A

Notifications endpoint

This endpoint is intended for working with notifications about events associated with resources.

Information about methods of the endpoint is provided in the table below.

Methods of the notice.Notice endpoint (kl.core.Notice interface)

Method	Method purpose and parameters	Potential danger of the method
`Create`	Purpose Creates a notification receiver. Parameters • [out] `notify` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the notification receiver. • [out] `rc` – return code.	Allows the kernel memory to be used up by creating a multitude of objects within it.
`SubscribeToObject`	Purpose Adds a "resource–event mask" entry to the notification receiver so that it can receive notifications about events associated with the defined resource and match the defined event mask. Parameters • [in] `notify` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the notification receiver. • [in] `object` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the resource. • [in] `evMask` – event mask. • [in] `evId` – ID of the "resource–event mask" entry. • [out] `rc` – return code.	Allows the kernel memory to be used up by creating a multitude of objects within it.
`UnsubscribeFromEvent`	Purpose Removes from the notification receiver "resource—event mask" entries with the specified identifier to prevent the receiver from getting notifications about events that match these entries. Parameters • [in] `notify` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the notification receiver. • [in] `evId` – ID of the "resource–event mask" entry. • [out] `rc` – return code.	N/A
`UnsubscribeFromObject`	Purpose Removes from the notification receiver "resource—event mask" entries that match the specified resource to prevent the receiver from getting notifications about events that match these entries. Parameters • [in] `notify` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the notification receiver.	N/A

	• [in] `object` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the resource. • [out] `rc` – return code.	
`GetEvent`	Purpose Extracts notifications from the receiver. Parameters • [in] `notify` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the notification receiver. • [in] `mdelay` – timeout for notifications to appear in the receiver, in milliseconds. • [out] `events` – sequence of notifications comprised of structures containing a "resource–event mask" entry ID, mask and counter of events associated with the resource. • [out] `rc` – return code.	N/A
`DropAndWake`	Purpose Removes from the specified notification receiver all "resource—event mask" entries, resumes all threads waiting for notifications to appear in the specified receiver; optionally prohibits adding of "resource—event mask" entries to the specified notification receiver. Parameters • [in] `notify` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the notification receiver. • [in] `finish` – value defining whether or not the addition of "resource–event mask" entries will be prohibited (`0` – will not be prohibited, `1` – will be prohibited). • [out] `rc` – return code.	N/A
`SetObjectEvent`	Purpose Signals the occurrence of events that are related to the defined user resource and match the defined event mask. Parameters • [in] `object` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the user resource. • [in] `evMask` – mask of events to be signaled. • [out] `rc` – return code.	N/A

IPC interrupt endpoint

This endpoint is intended for interrupting the `Call()` and `Recv()` locking system calls. (For example, this may be required to correctly terminate a process.)

Information about methods of the endpoint is provided in the table below.

Methods of the ipc.IPC endpoint (kl.core.IPC interface)

Method	Method purpose and parameters	Potential danger of the method
`CreateSyncObject`	Purpose Creates an IPC synchronization object. An IPC synchronization object is used to interrupt `Call()` and `Recv()` locking system calls in threads of the calling process. A `Call()` can be interrupted only when it is awaiting a `Recv()` call by the server. `Recv()` can be interrupted only when it is waiting to receive an IPC request from a client. The handle of an IPC synchronization object cannot be transferred to another process because the necessary flag for this operation is not set in the permissions mask of this handle. Parameters • [out] `syncHandle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the IPC synchronization object. • [out] `rc` – return code.	Allows the kernel memory to be used up by creating a multitude of objects within it.
`SetInterrupt`	Purpose Switches the defined IPC synchronization object to a state in which the `Call()` and `Recv()` system calls are interrupted. Parameters • [in] `syncHandle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the IPC synchronization object. • [out] `rc` – return code.	N/A
`ClearInterrupt`	Purpose Switches the defined IPC synchronization object to a state in which the `Call()` and `Recv()` system calls are not interrupted. Parameters • [in] `syncHandle` – value whose binary representation consists of multiple fields, including a handle field and a handle permissions mask field. The handle identifies the IPC synchronization object. • [out] `rc` – return code.	N/A

Hardware platform firmware communication endpoint

The endpoint is intended for interaction with embedded software (firmware) of the hardware platform via the EFI interface.

Information about methods of the endpoint is provided in the table below.

Methods of the efi.Efi endpoint (kl.core.Efi interface)

Method	Method purpose and parameters	Potential danger of the method
`GetTime`	Purpose Lets you get the current time from the time source on the hardware platform and the characteristics of this source.	N/A

	Parameters • [out] `time` – structure containing time information. • [out] `capabilities` – structure containing the characteristics of the time source. • [out] `rc` – return code.	
`SetTime`	Purpose Sets the current time in the time source on the hardware platform. Parameters • [in] `time` – structure containing time information. • [out] `rc` – return code.	Lets you set the current time in the time source on the hardware platform.
`GetVariable`	Purpose Lets you get the value of the EFI variable. Parameters • [in] `variableName` – sequence containing the name of a variable with a terminating zero. • [in] `vendorGuid` – structure containing the unique ID of the vendor related to the variable. • [out] `attributes` – value whose bits are interpreted as attributes of a variable. • [out] `dataSizeReal` – size (bytes) of data received via the `data` parameter. • [out] `data` – sequence containing the value of the variable. • [out] `rc` – return code.	Gets values of EFI variables that could contain critical system information.
`GetNextVariableName`	Purpose Enumerates the names of EFI variables. Parameters • [in] `variableNamePrev` – sequence containing the name of the variable with a terminating zero obtained during the previous call of the `GetNextVariableName` method, or null if the `GetNextVariableName` method is being called for the first time. • [in] `vendorGuidPrev` – structure containing the unique ID of the vendor related to the variable whose name was received during the previous call of the `GetNextVariableName` method, or null if the `GetNextVariableName` method is being called for the first time. • [out] `variableNameSizeReal` – size (bytes) of data received via the `variableNameNext` parameter. • [out] `variableNameNext` – sequence containing the name of a variable with a terminating zero. • [out] `vendorGuidNext` – structure containing the unique ID of the vendor related to the variable. • [out] `rc` – return code.	Gets the names of EFI variables that could contain critical system information.
`SetVariable`	Purpose Specifies the value of the EFI variable. Parameters	Lets you set the value of the EFI variable.

	• [in] `variableName` – sequence containing the name of a variable with a terminating zero. • [in] `vendorGuid` – structure containing the unique ID of the vendor related to the variable. • [in] `attributes` – value whose bits are interpreted as attributes of the variable. • [in] `data` – sequence containing the value of the variable. • [out] `rc` – return code.	
`QueryVariableInfo`	Purpose Gets information about EFI variables. Parameters • [in] `attributes` – value whose bits are interpreted as the types of those variables whose information you need to get. • [out] `maximumVariableStorageSize` – size (bytes) of memory that is intended for storing variables of specific types. • [out] `remainingVariableStorageSize` – size (bytes) of the unused portion of memory that is intended for storing variables of specific types. • [out] `maximumVariableSize` – maximum size (bytes) of variables of specific types. • [out] `rc` – return code.	N/A
`ResetSystem`	Purpose Restarts the hardware platform. Parameters • [in] `resetType` – value that defines the restart type, such as cold or hot restart. • [in] `resetStatus` – code that indicates the restart status, such as normal restart or restart due to malfunction. • [in] `resetData` – sequence containing information about the causes of restarts. • [out] `rc` – return code.	Lets you restart the hardware platform.

Using the system programs Klog and KlogStorage to perform a security audit

To perform a security audit, the system program `Klog` receives audit data from the KasperskyOS kernel by using the `libkos` library, decodes this data and forwards it via IPC to the system program `KlogStorage`, which acts as the server in this IPC interaction. The `KlogStorage` program sends audit data to standard output (or standard error) or saves it to a file by using VFS. The `KlogStorage` program can also forward file-written audit data to other programs via IPC.

The executable files of the `Klog` and `KlogStorage` programs are not provided in the KasperskyOS SDK. You will need to create them based on the provided static libraries.

Example of adding the system program Klog to a solution

Source code of the program

einit/src/klog_entity.c

```c
#include <klog/system_audit.h>
#include <klog_storage/client.h>
#include <ping/KlogEntity.edl.h>

int main(int argc, char *argv[])
{
    /* This function call creates a thread
     * that receives audit data from the kernel, decodes it and forwards it
     * via IPC to the KlogStorage program.
     * (The constant ping_KlogEntity_klog_audit_iid is defined in the header
     * file KlogEntity.edl.h, which contains the automatically generated
     * transport code.) */
    return klog_system_audit_run(KLOG_SERVER_CONNECTION_ID ":
                                 " KLOG_STORAGE_SERVER_CONNECTION_ID,
                                 ping_KlogEntity_klog_audit_iid);
}
```

Building a program

einit/CMakeLists.txt

```cmake
...
# Import Klog libraries from the
# KasperskyOS SDK
find_package (klog REQUIRED)
include_directories (${klog_INCLUDE})

# Generate transport code based on the formal specification of the
# Klog program
nk_build_edl_files (klog_edl_files
                    NK_MODULE "ping"
                    # The KlogEntity.edl file and other files
                    # in the formal specification of the Klog program
                    # are provided in the KasperskyOS SDK.
                    EDL "${RESOURCES}/edl/KlogEntity.edl")

# Create the executable file of the Klog program for the hardware platform
add_executable (KlogEntityHw "src/klog_entity.c")
target_link_libraries (KlogEntityHw ${klog_SYSTEM_AUDIT_LIB})
add_dependencies (KlogEntityHw klog_edl_files)

# Create the executable file of the Klog program for QEMU.
# (Identical to creating the executable file of the Klog program for
# the hardware platform, except for the build target name.
# Requires two build targets for the executable file of the
# Klog program with different names because the KLOG_ENTITY parameter of the
# CMake commands build_kos_hw_image() and build_kos_qemu_image()
# must specify different build targets.)
add_executable (KlogEntityQemu "src/klog_entity.c")
target_link_libraries (KlogEntityQemu ${klog_SYSTEM_AUDIT_LIB})
add_dependencies (KlogEntityQemu klog_edl_files)

# The Klog program does not need to be specified together with other programs
# to be included in the solution image. To include the Klog program
# in a solution, you must define the name of the build target for the executable file of this
# program via the KLOG_ENTITY parameter of the CMake commands
# build_kos_hw_image() and build_kos_qemu_image().
set (ENTITIES Client Server KlogStorageEntity FileVfs)
...
# The INIT_KlogEntity_PATH variable is used in the init.yaml.in file
# to define the name of the Klog program executable file. (The executable
# files of the Klog program for QEMU and for the hardware platform have
# different names that match the names of the build targets
# of these files by default.)
set (INIT_KlogEntity_PATH "KlogEntityHw")

# You must define the KLOG_ENTITY parameter
build_kos_hw_image (kos-image
                    EINIT_ENTITY EinitHw
                    ...
                    KLOG_ENTITY KlogEntityHw
```

```
                    IMAGE_FILES ${ENTITIES})

# The INIT_KlogEntity_PATH variable is used in the init.yaml.in file
# to define the name of the Klog program executable file. (The executable
# files of the Klog program for QEMU and for the hardware platform have
# different names that match the names of the build targets
# of these files by default.)
set (INIT_KlogEntity_PATH "KlogEntityQemu")

# You must define the KLOG_ENTITY parameter
build_kos_qemu_image (kos-qemu-image
                      EINIT_ENTITY EinitQemu
                      ...
                      KLOG_ENTITY KlogEntityQemu
                      IMAGE_FILES ${ENTITIES})
```

Program process dictionary in the init description template

einit/src/init.yaml.in

```
...
- name: ping.KlogEntity
  # The variable INIT_KlogEntity_PATH is defined in the file einit/CMakeLists.txt.
  path: @INIT_KlogEntity_PATH@
  connections:
  - target: ping.KlogStorageEntity
    id: {var: KLOG_STORAGE_SERVER_CONNECTION_ID, include: klog_storage/client.h}
...
```

Policy description for the program

einit/src/security.psl.in

```
...
use nk.base._
...
use EDL kl.core.Core
...
use EDL ping.KlogEntity
use EDL ping.KlogStorageEntity
...
use audit_profile._
use core._
...
/* Interaction with the KlogStorage program */

request dst=ping.KlogStorageEntity {
    match endpoint=klogStorage.storage {
        match method=write {
            match src=ping.KlogEntity { grant () }
        }
    }
}

response src=ping.KlogStorageEntity {
    match endpoint=klogStorage.storage {
        match method=write {
            match dst=ping.KlogEntity { grant () }
        }
    }
}

error src=ping.KlogStorageEntity {
    match endpoint=klogStorage.storage {
        match method=write {
            match dst=ping.KlogEntity { grant () }
        }
    }
}
...
```

einit/src/core.psl

```
...
/* Interaction with the kernel */

request dst=kl.core.Core {
    match endpoint=sync.Sync {
        match method=Wake {
            ...
            match src=ping.KlogEntity { grant () }
            ...
            }
        match method=Wait {
           ...
            match src=ping.KlogEntity { grant () }
          ...
            }
    }
    match endpoint=task.Task {
        match method=FreeSelfEnv {
            ...
            match src=ping.KlogEntity { grant () }
            ...
            }
        match method=GetPath {
            ...
            match src=ping.KlogEntity { grant () }
            ...
            }
        match method=GetName {
            ...
            match src=ping.KlogEntity { grant () }
            ...
            }
        match method=Exit {
            ...
            match src=ping.KlogEntity { grant () }
            ...
            }
    }
    match endpoint=vmm.VMM {
        match method=Allocate {
            ...
            match src=ping.KlogEntity { grant () }
            ...
            }
        match method=Commit {
            ...
            match src=ping.KlogEntity { grant () }
            ...
            }
        match method=Protect {
            ...
            match src=ping.KlogEntity { grant () }
            ...
            }
        match method=Free {
            ...
            match src=ping.KlogEntity { grant () }
            ...
            }
    }
    match endpoint=thread.Thread {
        match method=SetTls {
            ...
            match src=ping.KlogEntity { grant () }
            ...
            }
        match method=Create {
            ...
            match src=ping.KlogEntity { grant () }
            ...
            }
        match method=Resume {
            ...
            match src=ping.KlogEntity { grant () }
            ...
            }
        match method=Attach {
            ...
            match src=ping.KlogEntity { grant () }
            ...
```

```
            }
        match method=Exit {
            ...
            match src=ping.KlogEntity { grant () }
            ...
            }
        match method=GetSchedPolicy {
            ...
            match src=ping.KlogEntity { grant () }
            ...
            }
        match method=SetSchedPolicy {
            ...
            match src=ping.KlogEntity { grant () }
            ...
            }
    }
    match endpoint=hal.HAL {
        match method=GetEntropy {
            ...
            match src=ping.KlogEntity { grant () }
            ...
            }
        match method=DebugWrite {
            ...
            match src=ping.KlogEntity { grant () }
            ...
            }
        match method=GetEnv {
            ...
            match src=ping.KlogEntity { grant () }
            ...
            }
    }
    match endpoint=handle.Handle {
        match method=Close {
            ...
            match src=ping.KlogEntity { grant () }
            ...
            }
    }
    match endpoint=audit.Audit {
        match src=ping.KlogEntity { grant () }
        }
}

response src=kl.core.Core {
    ...
    match dst=ping.KlogEntity { grant () }
    ...
}

error src=kl.core.Core {
    ...
    match dst=ping.KlogEntity { grant () }
    ...
}
...
```

Example of adding the system program KlogStorage to a solution to forward audit data to standard error

Source code of the program

klog_storage/src/klog_storage_entity.c

```
#include <klog_storage/server.h>
#include <ping/KlogStorageEntity.edl.h>
#include <stdio.h>

/* Define the data type for a fictitious context.
 * Required for defining functions that implement
```

```
 * interface methods, and for dispatcher configuration. */
struct Context
{
    int some_data;
};

/* Define the function that forwards audit data to
 * standard error. (Use of the ctx parameter is not required, but a
 * void* type parameter must be the first parameter in the function signature to
 * match the type of pointer that is used by the dispatcher
 * to call this function.) */
static int _write(struct Context *ctx, const struct kl_KlogStorage_Entry *entry)
{
    fprintf(stderr, "%s\n", entry->msg);
    return 0;
}

/* Define a fictitious function for reading audit data.
 * (Required for dispatcher configuration to avoid errors
 * if the interface method for reading audit data is called.) */
static int _read_range(struct Context *ctx, nk_uint64_t first_id,
nk_uint64_t last_id, struct kl_KlogStorage_Entry *entries)
{
    return 0;
}

/* Define a fictitious function for reading audit data.
 * (Required for dispatcher configuration to avoid errors
 * if the interface method for reading audit data is called.) */
static int _read(struct Context *ctx, nk_uint32_t num_entries,
struct kl_KlogStorage_Entry *entries)
{
    return 0;
}

int main(int argc, char *argv[])
{
    /* Declaration of a fictitious context */
    static struct Context ctx;

    /* Configure the dispatcher so that when IPC requests
     * containing audit data are received from the Klog program, the dispatcher calls the function that forwards
     * this data to standard error. (The functions for reading audit data
     * and the context are fictitious.However, you can create your own
     * implementations of the _write(), _read() and _read_range() functions for working with
     * audit data storage.In this case, the context may be
     * used to store the storage state.) */
    struct kl_KlogStorage *iface =
    klog_storage_IKlog_storage_dispatcher(&ctx,
                                          (kl_KlogStorage_write_func)_write,
                                          (kl_KlogStorage_read_func)_read,
                                          (kl_KlogStorage_read_range_func)_read_range);
    struct kl_KlogStorage_component *comp =klog_storage_storage_component(iface);

    /* This function call starts the IPC request processing loop.
     * (The constants ping_KlogStorageEntity_klogStorage_iidOffset and
     * ping_KlogStorageEntity_klogStorage_storage_iid are defined in the header file
     * KlogStorageEntity.edl.h, which contains the automatically generated
     * transport code.) */
    return klog_storage_run(KLOG_STORAGE_SERVER_CONNECTION_ID,
                            ping_KlogStorageEntity_klogStorage_iidOffset,
                            ping_KlogStorageEntity_klogStorage_storage_iid,
                            comp);
}
```

Building a program

klog_storage/CMakeLists.txt

```
# Import KlogStorage libraries from the
# KasperskyOS SDK
find_package (klog_storage REQUIRED)
include_directories (${klog_storage_INCLUDE})

# Generate transport code based on the formal specification of the
# KlogStorage program
nk_build_edl_files (klog_storage_edl_files
```

```
                    NK_MODULE "ping"
                    # The KlogStorageEntity.edl file and other files
                    # in the formal specification of the KlogStorage program
                    # are provided in the KasperskyOS SDK.
                    EDL "${RESOURCES}/edl/KlogStorageEntity.edl")

# Create the executable file of the KlogStorage program
add_executable (KlogStorageEntity "src/klog_storage_entity.c")
target_link_libraries (KlogStorageEntity ${klog_storage_SERVER_LIB})
add_dependencies (KlogStorageEntity klog_edl_files klog_storage_edl_files)
```

Program process dictionary in the init description template

einit/src/init.yaml.in

```
...
- name: ping.KlogStorageEntity
...
```

Policy description for the program

einit/src/security.psl.in

```
...
use nk.base._
...
use EDL kl.core.Core
...
use EDL ping.KlogEntity
use EDL ping.KlogStorageEntity
...
use audit_profile._
use core._
...
/* Interaction with the Klog program */

request dst=ping.KlogStorageEntity {
    match endpoint=klogStorage.storage {
        match method=write {
            match src=ping.KlogEntity { grant () }
        }
    }
}

response src=ping.KlogStorageEntity {
    match endpoint=klogStorage.storage {
        match method=write {
            match dst=ping.KlogEntity { grant () }
        }
    }
}

error src=ping.KlogStorageEntity {
    match endpoint=klogStorage.storage {
        match method=write {
            match dst=ping.KlogEntity { grant () }
        }
    }
}
...
```

einit/src/core.psl

```
...
/* Interaction with the kernel */

request dst=kl.core.Core {
    match endpoint=sync.Sync {
        match method=Wake {
            ...
```

```
        match src=ping.KlogStorageEntity { grant () }
        ...
        }
    match method=Wait {
        ...
        match src=ping.KlogStorageEntity { grant () }
        ...
        }
}
match endpoint=task.Task {
    match method=FreeSelfEnv {
        ...
        match src=ping.KlogStorageEntity { grant () }
        ...
        }
    match method=GetPath {
        ...
        match src=ping.KlogStorageEntity { grant () }
        ...
        }
    match method=GetName {
        ...
        match src=ping.KlogStorageEntity { grant () }
        ...
        }
    match method=Exit {
        ...
        match src=ping.KlogStorageEntity { grant () }
        ...
        }
}
match endpoint=vmm.VMM {
    match method=Allocate {
        ...
        match src=ping.KlogStorageEntity { grant () }
        ...
        }
    match method=Commit {
        ...
        match src=ping.KlogStorageEntity { grant () }
        ...
        }
    match method=Protect {
        ...
        match src=ping.KlogStorageEntity { grant () }
        ...
        }
    match method=Free {
        ...
        match src=ping.KlogStorageEntity { grant () }
        ...
        }
}
match endpoint=thread.Thread {
    match method=SetTls {
        ...
        match src=ping.KlogStorageEntity { grant () }
        ...
        }
    match method=Create {
        ...
        match src=ping.KlogStorageEntity { grant () }
        ...
        }
    match method=Resume {
        ...
        match src=ping.KlogStorageEntity { grant () }
        ...
        }
}
match endpoint=hal.HAL {
    match method=GetEntropy {
        ...
        match src=ping.KlogStorageEntity { grant () }
        ...
        }
    match method=DebugWrite {
        ...
        match src=ping.KlogStorageEntity { grant () }
        ...
        }
    match method=GetEnv {
```

```
            ...
            match src=ping.KlogStorageEntity { grant () }
            ...
            }
    }
    match endpoint=handle.Handle {
        match method=Close {
            ...
            match src=ping.KlogStorageEntity { grant () }
            ...
            }
    }
}

response src=kl.core.Core {
    ...
    match dst=ping.KlogStorageEntity { grant () }
    ...
}

error src=kl.core.Core {
    ...
    match dst=ping.KlogStorageEntity { grant () }
    ...
}
...
```

Example of adding the system program KlogStorage to a solution to write audit data to a file

Source code of the program

klog_storage/src/klog_storage_entity.c

```
#include <klog_storage/server.h>
#include <klog_storage/file_storage.h>
#include <ping/KlogStorageEntity.edl.h>

int main(int argc, char *argv[])
{
    /* This function call starts the IPC request processing loop.
     * The audit data will be written to the file /etc/klog_storage.log, which can
     * hold no more than 100 entries. When the file is completely full, the previous
     * entries will be replaced by new entries starting at the beginning of the file. If the last parameter
     * of the function has a value other than 1, the KlogStorage program at startup
     * opens the existing file and begins to write audit data at the specific position
     * that was set in the file after the previous write operation. If the last
     * parameter of the function has a value of 1, a new empty file will be created.
     * (The constants ping_KlogStorageEntity_klogStorage_iidOffset and
     * ping_KlogStorageEntity_klogStorage_storage_iid are defined in the header
     * file KlogStorageEntity.edl.h, which contains the automatically generated
     * transport code.) */
    return klog_storage_file_storage_run(KLOG_STORAGE_SERVER_CONNECTION_ID,
                                         "/etc/klog_storage.log",
                                         ping_KlogStorageEntity_klogStorage_iidOffset,
                                         ping_KlogStorageEntity_klogStorage_storage_iid,
                                         100,
                                         0);
}
```

Building a program

The difference between the `CMake` commands for building the `KlogStorage` program that writes audit data to a file and the `CMake` commands for building the version of this program that sends audit data to standard error comprises the following modification:

klog_storage/CMakeLists.txt

```
...
# When creating the executable file of the KlogStorage program, you must
# link it to the klog_storage_file_storage library.
target_link_libraries (KlogStorageEntity ${klog_storage_FILE_STORAGE_LIB})
...
```

Program process dictionary in the init description template

einit/src/init.yaml.in

```
...
- name: ping.KlogStorageEntity
  connections:
  - target: file_vfs.FileVfs
    id: {var: _VFS_CONNECTION_ID, include: vfs/defs.h}
...
```

Security policy description for the program

The difference between a policy description for a `KlogStorage` program that writes audit data to a file and a policy description for a version of this program that sends audit data to standard error comprises the following addition:

einit/src/security.psl.in

```
...
use EDL file_vfs.FileVfs
...
use vfs._
...
```

einit/src/vfs.psl

```
...
/* Interaction with the VFS program */

request dst=file_vfs.FileVfs {
    match src=ping.KlogStorageEntity { grant () }
}

response src=file_vfs.FileVfs {
    match dst=ping.KlogStorageEntity { grant () }
}

error src=file_vfs.FileVfs {
    match dst=ping.KlogStorageEntity { grant () }
}
...
```

Forwarding audit data to other programs

To forward file-written audit data via IPC, the `KlogStorage` program provides the `read` and `readRange` interface methods defined in the file `sysroot-*-kos/include/kl/KlogStorage.idl` from the KasperskyOS SDK.

The executable file of the program that needs to receive the audit data must be linked to the client library of the `KlogStorage` program:

klog_reader/CMakeLists.txt

```
# Import KlogStorage libraries from the
# KasperskyOS SDK
find_package (klog_storage REQUIRED)
include_directories (${klog_storage_INCLUDE})
...
# Create the executable file of the program that needs to
# receive audit data from the KlogStorage program.
add_executable (KlogReader "src/klog_reader.c")
target_link_libraries (KlogReader ${klog_storage_CLIENT_LIB})
...
```

Source code for receiving audit data from the `KlogStorage` program:

klog_reader/src/klog_reader.c

```
#include <klog_storage/client.h>
...
int main(int argc, char *argv[])
{
...
    struct Klog_storage_ctx *storage =
    klog_storage_init(KLOG_STORAGE_SERVER_CONNECTION_ID);

    struct kl_KlogStorage_Entry first_entries[10], latest_entries [10];

    /* Read the first ten entries */
    int f_count = klog_storage_read_range(klog_storage_IKlog_storage(storage),
                                          1,
                                          10,
                                          first_entries);

    /* Read the last ten entries */
    int l_count = klog_storage_read(klog_storage_IKlog_storage(storage),
                                    10,
                                    latest_entries);
...
}
```

Security patterns for development under KasperskyOS

Each KasperskyOS-based solution has specific usage scenarios and is designed to counteract specific security threats. Nonetheless, there are some typical scenarios and threats encountered in many different solutions. This section describes the typical risks and threats, and contains a description of architectural patterns that can be employed to increase the security of a solution.

A *security pattern (or template)* describes a specific recurring security issue that arises in certain known contexts, and provides a well-proven, general scheme for resolving this kind of security issue. A pattern is not a finished project that can be converted directly into code. Instead, it is a solution to a general problem encountered in various projects.

A *security pattern system* is a set of security patterns together with instructions on their implementation, combination, and practical use when designing secure software systems.

Security patterns resolve security issues at different levels, beginning with patterns at the architectural level, including high-level design of the system, and ending with implementation-level patterns that contain recommendations on how to implement functions or methods.

This section describes the set of security patterns whose implementation examples are provided in KasperskyOS Community Edition.

> Security patterns are described in a multitude of information security resources. Each pattern is accompanied by a list of the resources that were used to prepare its description.

Distrustful Decomposition pattern

Description

When using a monolithic application, a single process must be granted all the privileges necessary for the application to operate. This issue is resolved by the `Distrustful Decomposition` pattern.

The purpose of the `Distrustful Decomposition` pattern is to divide application functionality among individual processes that require different levels of privileges, and to control the interaction between these processes instead of creating a monolithic application.

Using the `Distrustful Decomposition` pattern reduces the following:

- Attack surface for each process.
- Functionality and data that a hacker will be able to access if one of the processes is compromised.

Alternate names

`Privilege Reduction`.

Context

Different functions of an application require different levels of privileges.

Problem

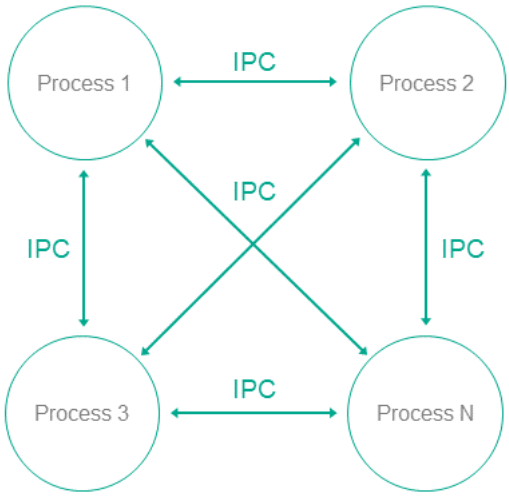
An unsophisticated implementation of an application combines many functions requiring different privileges into one component. This component would need to be run with the maximum level of privileges required for any one of these many functions.

Solution

The `Distrustful Decomposition` pattern divides functionality among individual processes and isolates potential vulnerabilities within a small subset of the system. A cybercriminal who conducts a successful attack will be able to use only the functionality and data of a single compromised component instead of the entire application.

Structure

This pattern divides one monolithic application into multiple applications that are run as individual processes that could potentially have different privileges. Each process implements a small, clearly defined set of functions of the application. Processes use interprocess communication mechanism to exchange data.



Operation

- In KasperskyOS, an application is divided into processes.
- Processes can exchange messages via IPC.
- A user or remote system connects to the process that provides the necessary functionality with the level of privileges sufficient to perform the requested functions.

Implementation recommendations

Interaction between processes can be unidirectional or bidirectional. It is recommended to always use unidirectional interaction whenever possible. Otherwise, the potential attack surface of individual components increases, which reduces the overall security of the entire system. If bidirectional IPC is used, processes should not trust bidirectional data exchange. For example, if a file system is used for IPC, file contents cannot be trusted.

Specialized implementation in KasperskyOS

In universal operating systems such as Linux or Windows, this pattern does not use anything except the standard process/privileges model that already exists in these operating systems. Each program is run in its own process space with potentially different privileges of the specific user in each process. However, an attack on the OS kernel would reduce the effectiveness of this pattern.

Use of this pattern when developing for KasperskyOS means that control over processes and IPC is entrusted to the microkernel, which is difficult to successfully attack. The Kaspersky Security Module is used for IPC control.

Use of KasperskyOS mechanisms ensures a high level of reliability of the software system with the same or less effort required from the developer when compared to the use of this pattern in programs running under universal operating systems.

In addition, KasperskyOS provides the capability for flexible configuration of security policies. Moreover, the process of defining and editing security policies is potentially independent of the process of developing the applications.

Linked patterns

Use of the `Distrustful Decomposition` pattern involves use of the Defer to Kernel and Policy Decision Point patterns.

Implementation examples

Examples of an implementation of the `Distrustful Decomposition` pattern:

- Secure Logger
- Separate Storage

Sources of information

The `Distrustful Decomposition` pattern is described in detail in the following resources:

- Chad Dougherty, Kirk Sayre, Robert C. Seacord, David Svoboda, Kazuya Togashi (JPCERT/CC), "Secure Design Patterns" (March-October 2009). Software Engineering Institute. https://resources.sei.cmu.edu/asset_files/TechnicalReport/2009_005_001_15110.pdf
- Dangler, Jeremiah Y., "Categorization of Security Design Patterns" (2013). Electronic Theses and Dissertations. Paper 1119. https://dc.etsu.edu/etd/1119

Secure Logger example

The `Secure Logger` example demonstrates use of the Distrustful Decomposition pattern for separating event log read/write functionality.

Example architecture

The security goal of the `Secure Logger` example is to prevent any possibility of distortion or deletion of information from the event log. This example utilizes the capabilities provided by KasperskyOS to achieve this security goal.

A logging system can be examined by distinguishing the following functional steps:

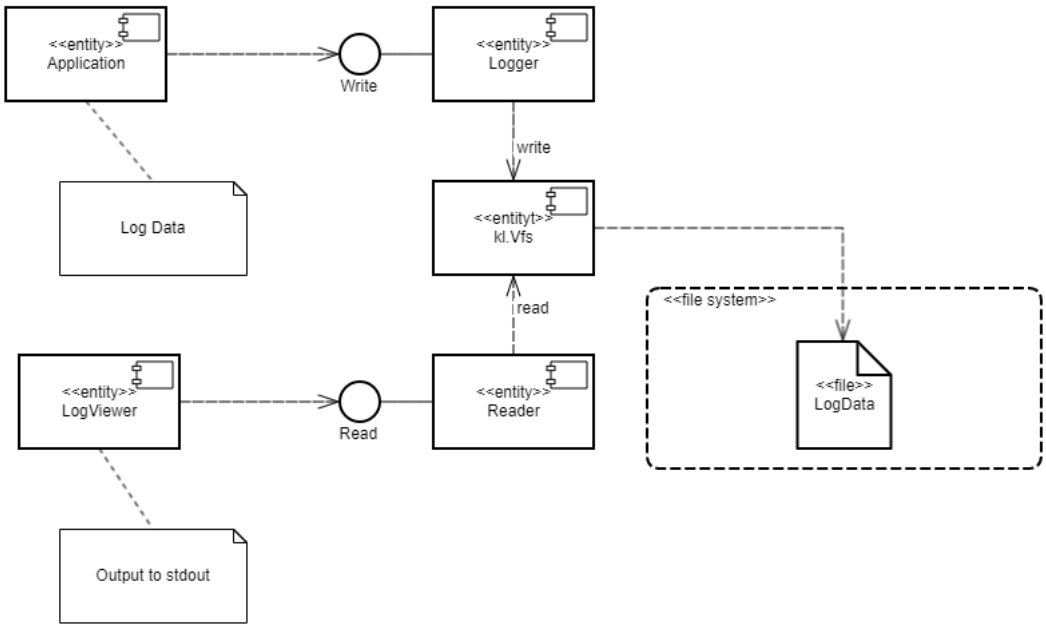
- Generate information to be written to the log.
- Save information to the log.
- Read entries from the log.
- Provide entries in a convenient format for the consumer.

Accordingly, the logging subsystem can be divided into four processes depending on the required functional capabilities of each process.

For this purpose, the `Secure Logger` example contains the following four programs: `Application`, `Logger`, `Reader` and `LogViewer`.

- The `Application` program initiates the creation of entries in the event log maintained by the `Logger` program.
- The `Logger` program creates entries in the log and writes them to the disk.
- The `Reader` program reads entries from the disk to send them to the `LogViewer` program.
- The `LogViewer` program sends entries to the user.

The IPC interface provided by the `Logger` program is intended *only* for writing to storage. The IPC interface of the `Reader` program is intended only for reading from storage. The example architecture looks as follows:



- The `Application` program uses the interface of the `Logger` program to save log entries.
- The `LogViewer` program uses the interface of the `Reader` program to read the log entries and present them to a user.

The `LogViewer` program normally has external channels for interacting with a user (for example, to receive data write commands and to provide data to a user). Naturally, this program is an untrusted component of the system, and therefore could potentially be used to conduct an attack. However, even if a successful attack results in the infiltration of unauthorized executable code into the `LogViewer` program, information in the log cannot be distorted through this program. This is because the program can only utilize the data read interface, which cannot actually be used to distort or delete data. Moreover, the `LogViewer` program does not have the capability to gain access to other interfaces because this access is controlled by the security module.

A security policy in the `Secure Logger` example has the following characteristics:

- The `Application` program has the capability to query the `Logger` program to create a new entry in the event log.
- The `LogViewer` program has the capability to query the `Reader` program to read entries from the event log.
- The `Application` program *does not* have the capability to query the `Reader` program to read entries from the event log.
- The `LogViewer` program *does not* have the capability to query the `Logger` program to create a new entry in the event log.

Example files

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

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

Building and running example

See Building and running examples section.

Separate Storage example

The `Separate Storage` example demonstrates use of the Distrustful Decomposition pattern to separate data storage for trusted and untrusted applications.

Example architecture

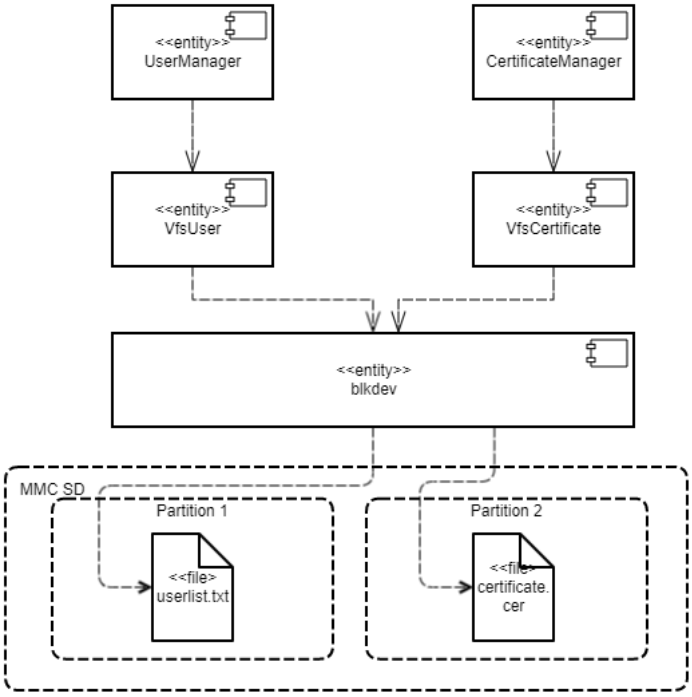
The `Separate Storage` example contains two user programs: `UserManager` and `CertificateManager`.

These programs work with data located in the corresponding files:

- The `UserManager` program works with data from the `userlist.txt` file.

- The `CertificateManager` program works with data from the `certificate.cer` file.

Each of these programs uses its own instance of the VFS program to access a separate file system. Each VFS program includes a block device driver linked to an individual logical drive partition. The `UserManager` program does not have access to the file system of the `CertificateManager` program, and vice versa.



This architecture guarantees that if there is an attack or error in any of the `UserManager` or `CertificateManager` programs, this program will not be able to access any file that was not intended for the specific program's operations.

A security policy in the `Separate Storage` example has the following characteristics:

- The `UserManager` program has access to the file system *only* through the `VfsUser` program.
- The `CertificateManager` program has access to the file system *only* through the `VfsCertificate` program.

Example files

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

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

Building and running example

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

```
$ cd build/einit
# Before running the following command, be sure that the path to
# the directory with the qemu-system-aarch64 executable file is saved in
# the PATH environment variable. If it is not there,
# add it to the PATH variable.
```

```
$ qemu-system-aarch64 -m 2048 -machine vexpress-a15 -nographic -monitor none -sd sdcard0.img -kernel kos-qemu-image
```

See also Building and running examples section.

Preparing an SD card to run on Raspberry Pi 4 B

To run the `Separate Storage` example on Raspberry Pi 4 B, the following additional actions are necessary:

- Create a `/lib` directory in the SD card boot sector unless one already exists.
- Copy the contents of the `build/hdd/part1/lib` directory that was generated while building the example to the `/lib` directory in the SD card boot sector.
- The SD card must contain both a bootable partition with the solution image as well as 2 additional partitions with the `ext2` or `ext3` file systems.
- The first additional partition must contain the `userlist.txt` file from the `./resources/files/` directory.
- The second additional partition must contain the `certificate.cer` file from the `./resources/files/` directory.

> To run the `Separate Storage` example on Raspberry Pi 4 B, you can use an SD card prepared for running the `vfs_extfs` example on Raspberry Pi 4 B after copying the `userlist.txt` and `certificate.cer` files to the appropriate partitions.

Defer to Kernel pattern

Description

The `Defer to Kernel` pattern takes advantage of permission control at the OS kernel level.

The purpose of this pattern is to utilize mechanisms available at the OS kernel level to clearly separate the functionality requiring elevated privileges from the functionality that does not require elevated privileges. By using kernel mechanisms, we do not have to implement new tools for arbitrating security decisions at the user level.

Alternate names

`Policy Enforcement Point (PEP)`, `Protected System`, `Enclave`.

Context

The `Defer to Kernel` pattern is applicable if the system has the following characteristics:

- The system has processes that run without elevated privileges, including user processes.

- Some system functions require elevated privileges that must be verified before processes are granted access to data.
- You need to verify not only the privileges of the requesting process, but also the overall permissibility of the requested operation within the operational context of the entire system and its overall security.

Problem

When functionality is divided among various processes with different levels of privileges, these privileges must be verified when a request is made from one process to another. These verifications must be carried out and their resulting permissions must be granted by trusted code that has a minimal risk of being compromised. The trustworthiness of application code is almost always questionable due to its sheer volume and due to its primary orientation toward implementation of functional requirements.

Solution

Clearly separate privileged functionality and data from non-privileged functionality and data at the process level, and give the OS kernel control of interprocess communication (IPC), including verification of access rights when there is a request for functionality or data requiring elevated privileges, and verification of the overall state of the system and the states of individual processes at the time of the request.

Structure

Operation

- Functionality and management of data with various privileges are compartmentalized among processes.
- The OS kernel ensures isolation of processes.
- `Process-1` wants to request privileged functionality or data from `Process-2` using IPC.
- The kernel controls IPC and allows or denies communication based on security policies and based on the available information regarding the operational context and state of `Process-1`.

Implementation recommendations

To ensure that a specific implementation of a pattern operates securely and reliably, the following is required:

- **Isolation**

  Complete and guaranteed isolation of processes must be ensured.

- **Inability to bypass the kernel**

  Absolutely all IPC interactions must be controlled by the kernel.

- **Kernel self-defense**

  The trustworthiness of the kernel must be ensured through its own means of protection against compromise.

- **Provability**

  The kernel requires a certain level of guaranteed security and reliability.

- **Capability for external computation of access permissions**

  Access permissions must be computed at the OS level, and must not be implemented in application code.

  For this purpose, tools must be provided for describing access policies so that security policies are detached from the business logic.

Specialized implementation in KasperskyOS

The KasperskyOS kernel guarantees isolation of processes and serves as a Policy Enforcement Point (PEP).

Linked patterns

The `Defer to Kernel` pattern is a special case of the Distrustful Decomposition and Policy Decision Point patterns. The `Policy Decision Point` pattern defines the abstraction process that intercepts all requests to resources and verifies that they comply with the defined security policy. The distinctive feature of the `Defer to Kernel` pattern is that the verification process is performed by the OS kernel, which is a more reliable and portable solution that reduces the time spent on development and testing.

Impacts

By making the OS kernel responsible for applying the access policy, you separate the security policy from the business logic (which may be very complicated) and thereby simplify development and improve portability through the use of OS kernel functions.

This also makes it possible to prove the overall security of a solution by simply demonstrating that the kernel is operating correctly. The difficulty in proving correct execution of code grows nonlinearly as the size of the code increases. The `Defer to Kernel` pattern minimizes the amount of trusted code, provided that the OS kernel itself is not too large.

Implementation examples

Example of a `Defer to Kernel` pattern implementation: Defer to Kernel example.

Sources of information

The `Defer to Kernel` pattern is described in detail in the following resources:

- Chad Dougherty, Kirk Sayre, Robert C. Seacord, David Svoboda, Kazuya Togashi (JPCERT/CC), "Secure Design Patterns" (March-October 2009). Software Engineering Institute. https://resources.sei.cmu.edu/asset_files/TechnicalReport/2009_005_001_15110.pdf

- Dangler, Jeremiah Y., "Categorization of Security Design Patterns" (2013). Electronic Theses and Dissertations. Paper 1119. https://dc.etsu.edu/etd/1119

- Schumacher, Markus, Fernandez-Buglioni, Eduardo, Hybertson, Duane, Buschmann, Frank, and Sommerlad, Peter. "Security Patterns: Integrating Security and Systems Engineering" (2006)

Defer to Kernel example

The `Defer to Kernel` example demonstrates the use of Defer to Kernel and Policy Decision Point patterns.

The `Defer to Kernel` example contains three user programs: `PictureManager`, `ValidPictureClient` and `NonValidPictureClient`.

In this example, the `ValidPictureClient` and `NonValidPictureClient` programs query the `PictureManager` program to receive information.

Only the `ValidPictureClient` program is allowed to interact with the `PictureManager` program.

The KasperskyOS kernel guarantees isolation of running programs (processes).

Control of interaction between programs in KasperskyOS is delegated to the Kaspersky Security Module. The subsystem analyzes each sent request and response and decides whether to allow or deny delivery based on the defined security policy.

A security policy in the `Defer to Kernel` example has the following characteristics:

- The `ValidPictureClient` program is explicitly allowed to interact with the `PictureManager` program.

- The `NonValidPictureClient` program is explicitly *not* allowed to interact with the `PictureManager` program. This means that this interaction is denied (based on the *Default Deny principle*).

Dynamically created IPC channels

The example also demonstrates the capability to dynamically create IPC channels between processes. IPC channels are dynamically created by using a name server, which is a specialized core endpoint provided by the `NameServer` program. The capability to dynamically create IPC channels allows you to change the topology of interaction between programs on the fly.

Any program that is allowed to interact with `NameServer` via IPC can register its own interfaces in the name server. Another program can request the registered interfaces from the name server, and then connect to the relevant interface.

The security module is used to control interactions via IPC (even those that were created dynamically).

Example files

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

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

Building and running example

See Building and running examples section.

Policy Decision Point pattern

Description

The `Policy Decision Point` pattern encapsulates the computation of decisions based on security model methods into a separate system component that ensures that these security methods are performed in their full scope and correct sequence.

Alternate names

`Check Point`, `Access Decision Function`.

Context

The system has functions with different levels of privileges, and the security policy is complex (contains many security model methods bound to security events).

Problem

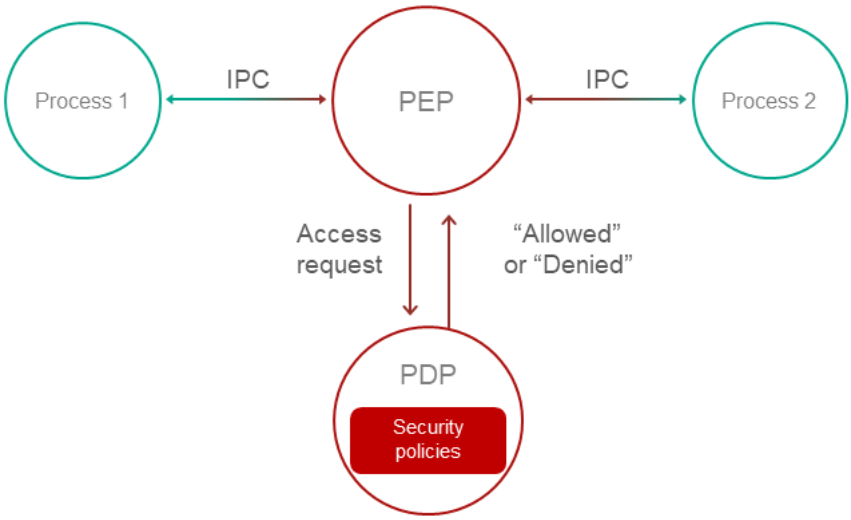
If security policy checks are divided among different system components, the following issues arise:

- You have to carefully make sure that all necessary checks are performed in all required cases.
- It is difficult to ensure that all checks are performed in the correct order.
- It is difficult to prove that the verification system is operating correctly, has no conflicts, and its integrity has not been compromised.
- The security policy is linked to the business logic. This means that any modification of the security policy requires changes to the business logic, which complicates support and increases the likelihood of errors.

Solution

All verifications of security policy compliance are conducted in a separate component called a Policy Decision Point (PDP). This component is responsible for ensuring that verifications are conducted in their correct sequence and scope. Policy checks are separated from the code that implements the business logic.

Structure



Operation

- A Policy Enforcement Point (PEP) receives a request to access functionality or data.

  For example, the PEP may be the OS kernel. For more details, refer to Defer to Kernel pattern.

- The PEP gathers the request attributes required for making decisions on access control.
- The PEP requests an access control decision from the Policy Decision Point (PDP).
- The PDP computes a decision on whether to grant access based on the security policy and based on the information received in the request from the PEP.
- The PEP denies or allows interaction based on the decision of the PDP.

Implementation recommendations

Implementations must take into account the problem of "Verification time vs. Usage time". For example, if a security policy depends on the quickly changing status of a specific system object, a computed decision loses its relevance as quickly as the status changes. In a system that utilizes the **Policy Decision Point** pattern, you must take care to minimize the time interval between the access decision and the time when the request based on this decision is fulfilled.

Specialized implementation in KasperskyOS

The KasperskyOS kernel guarantees isolation of processes and serves as a Policy Enforcement Point (PEP).

Control of interaction between processes in KasperskyOS is delegated to the Kaspersky Security Module. This module analyzes each sent request and response and decides whether to allow or deny delivery based on the defined security policy. Therefore, the Kaspersky Security Module performs the role of the Policy Decision Point (PDP).

Impacts

This pattern configures a security policy without making any modifications to the code that implements the business logic, and delegates system support involving information security.

Linked patterns

Use of the `Policy Decision Point` pattern involves use of the Distrustful Decomposition and Defer to Kernel patterns.

Implementation examples

Example of a `Policy Decision Point` pattern implementation: Defer to Kernel example.

Sources of information

The `Policy Decision Point` pattern is described in detail in the following resources:

- Chad Dougherty, Kirk Sayre, Robert C. Seacord, David Svoboda, Kazuya Togashi (JPCERT/CC), "Secure Design Patterns" (March-October 2009). Software Engineering Institute. https://resources.sei.cmu.edu/asset_files/TechnicalReport/2009_005_001_15110.pdf
- Dangler, Jeremiah Y., "Categorization of Security Design Patterns" (2013). Electronic Theses and Dissertations. Paper 1119. https://dc.etsu.edu/etd/1119
- Schumacher, Markus, Fernandez-Buglioni, Eduardo, Hybertson, Duane, Buschmann, Frank, and Sommerlad, Peter. "Security Patterns: Integrating Security and Systems Engineering" (2006)
- Bob Blakley, Craig Heath, and members of The Open Group Security Forum. "Security Design Patterns" (April 2004). The Open Group. https://pubs.opengroup.org/onlinepubs/9299969899/toc.pdf

Privilege Separation pattern

Description

The `Privilege Separation` pattern involves the use of non-privileged isolated system modules for interaction with clients (other modules or users) that do not have any privileges. The purpose of the `Privilege Separation` pattern is to reduce the amount of code that is executed with special privileges without impacting or restricting application functionality.

The `Privilege Separation` pattern is a special case of the Distrustful Decomposition pattern.

Example

An unauthenticated user connects to a system that has functions requiring elevated privileges.

Context

The system has components with a large attack surface due to their high number of connections with unsafe sources and/or a complicated, potentially error-prone implementation.

Problem

When a client with unknown privileges interacts with a privileged component of the system, there are risks that the data and functionality accessible to that component could be compromised.

Solution

Interactions with unsafe clients must be conducted only through specially allocated components that have no privileges. The `Privilege Separation` pattern does not modify system functionality. Instead, it merely separates functionality into components with different privileges.

Operation

Pattern operations can be divided into two phases:

- **Pre-Authentication.** The client is not yet authenticated. It sends a request to a privileged primary process. The primary process creates a child process with no privileges (and no access to the file system). This child process performs client authentication.
- **Post-Authentication.** The client is authenticated and authorized. The privileged primary process creates a new child process that has privileges corresponding to the permissions of the client. This process is responsible for all subsequent interaction with the client.

Recommendations on implementation in KasperskyOS

At the **Pre-Authentication** phase, the primary process can save the state of each non-privileged process in the form of a finite-state machine and change the state of the finite-state machine during authentication.

Requests from child processes to the primary process are performed using standard IPC mechanisms. However, interaction control is conducted using the Kaspersky Security Module.

Impacts

If attackers gain control of a non-privileged process, they will not gain access to any privileged functions or data. If attackers gain control of an authorized process, they will obtain only the privileges of this process.

In addition, code that is organized in this manner is easier to check and test. You just have to pay special attention to the functionality that operates with elevated privileges.

Implementation examples

Example of a `Privilege Separation` pattern implementation: Device Access example.

Sources of information

The `Privilege Separation` pattern is described in detail in the following resources:

- Chad Dougherty, Kirk Sayre, Robert C. Seacord, David Svoboda, Kazuya Togashi (JPCERT/CC), "Secure Design Patterns" (March-October 2009). Software Engineering Institute.

https://resources.sei.cmu.edu/asset_files/TechnicalReport/2009_005_001_15110.pdf

- Dangler, Jeremiah Y., "Categorization of Security Design Patterns" (2013). Electronic Theses and Dissertations. Paper 1119. https://dc.etsu.edu/etd/1119

Device Access example

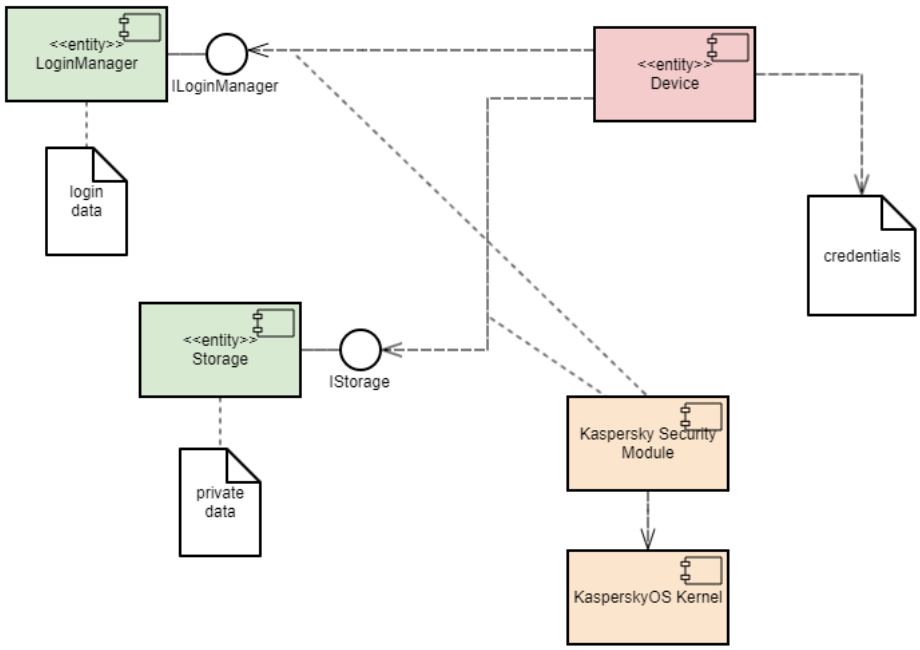
The `Device Access` example demonstrates use of the Privilege Separation pattern.

Example architecture

The example contains the following three programs: `Device`, `LoginManager` and `Storage`.

In this example, the `Device` program queries the `Storage` program to receive information and queries the `LoginManager` program for authorization.

The `Device` program obtains access to the `Storage` program after successful authorization.



This example demonstrates the capability to separate the authorization logic and the data access logic into independent components. This separation guarantees that data access can be opened only after successful authorization. The security module monitors whether authorization was successfully completed. This architecture also enables independent development and testing of the authorization logic and the data access provision logic.

A security policy in the `Device Access` example has the following characteristics:

- The `Device` program has the capability to query the `LoginManager` program for authorization.
- Calls of the `GetInfo()` method of the `Storage` program are managed by methods of the Flow security model:
  - The finite-state machine described in the `session` object configuration has two states: `unauthenticated` and `authenticated`.
  - The initial state is `unauthenticated`.

- Only transitions from `unauthenticated` to `authenticated` and vice versa are allowed.
- The `session` object is created when the `Device` program is started.
- When the `Device` program successfully calls the `Login()` method of the `LoginManager` program, the state of the `session` object changes to `authenticated`.
- When the `Device` program successfully calls the `Logout()` method of the `LoginManager` program, the state of the `session` object changes to `unauthenticated`.
- When the `Device` program calls the `GetInfo()` method of the `Storage` program, the current state of the `session` object is verified. The call is allowed only if the current state of the object is `authenticated`.

Example files

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

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

Building and running example

See Building and running examples section.

Information Obscurity pattern

Description

The purpose of the `Information Obscurity` pattern is to encrypt confidential data in otherwise unsafe environments and thereby protect against data theft.

Context

This pattern should be used when data is frequently transferred between parts of a system and/or between the system and other (external) systems.

Problem

Confidential data may be transmitted through an untrusted environment within one system (through untrusted components) or between different systems (through untrusted networks). If this environment is compromised, confidential data could be intercepted by a cybercriminal.

Solution

Data must be separated based on its specific level of confidentiality so that you can determine which data should be encrypted and which encryption algorithms should be used. Encryption and decryption may take a lot of time, therefore their use should be limited whenever possible. The `Information Obscurity` pattern resolves this issue by utilizing a specific confidentiality level to determine what exactly must be concealed with encryption.

Implementation examples

Example of an `Information Obscurity` pattern implementation: Secure Login example.

Sources of information

The `Information Obscurity` pattern is described in detail in the following resources:

- Dangler, Jeremiah Y., "Categorization of Security Design Patterns" (2013). Electronic Theses and Dissertations. Paper 1119. https://dc.etsu.edu/etd/1119
- Schumacher, Markus, Fernandez-Buglioni, Eduardo, Hybertson, Duane, Buschmann, Frank, and Sommerlad, Peter. "Security Patterns: Integrating Security and Systems Engineering" (2006)

Secure Login (Civetweb, TLS-terminator) example

The `Secure Login` example demonstrates use of the Information Obscurity pattern. This example demonstrates the capability to transmit critical system information through an untrusted environment.

Example architecture

This example simulates the acquisition of remote access to an IoT device by sending user account credentials (user name and password) to this device. The untrusted environment within the IoT device is the web server that responds to requests from users. Practical experience has shown that this kind of web server is easy to detect and frequently attacked successfully because IoT devices do not have built-in tools for protection against intrusion and other attacks. Users also gain access to the IoT device through an untrusted network. Obviously, encryption algorithms must be used in these types of conditions to protect user account credentials from being compromised.

In terms of the architecture in these systems, the following objects can be distinguished:

- Data source: user's browser.
- Point of communication with the device: web server.
- Subsystem for processing information from the user: authentication subsystem.

To employ cryptographic protection, the following steps must be completed:

1. Configure interaction between the data source and the device over the HTTPS protocol. This helps prevent unauthorized surveillance of HTTP traffic and MITM (man-in-the-middle) attacks.
2. Generate a shared secret between the data source and the information processing subsystem.

3. Use this secret to encrypt information on the data source side and to decrypt the information on the information processing subsystem side. This helps prevent data within the device from being compromised (at the point of communication).

The `Secure Login` example includes the following components:

- `Civetweb` web server (untrusted component, `WebServer` program).
- User authentication subsystem (trusted component, `AuthService` program).
- TLS terminator (trusted component, `TlsEntity` program). This component supports the TLS (transport layer security) mechanism. Together with the web server, the TLS terminator supports the HTTPS protocol on the device side (the web server interacts with the browser through the TLS terminator).

The user authentication process occurs as follows:

1. Using their browser, the user opens the page at `https://localhost:1106` (when running the example on QEMU) or at `https://<Raspberry Pi IP address>:1106` (when running the example on Raspberry Pi 4 B). HTTP traffic between the browser and TLS terminator will be transmitted in encrypted form, but the web server will work only with unencrypted HTTP traffic.

   > This example uses a self-signed certificate, so most up-to-date browsers will warn you that the connection is not secure. You need to agree to use this "insecure" connection, which will actually be encrypted despite the warning. When using certain browsers, the message `"mbedtlsErrorCode=-0x7780, Error performing SSL handshake: SSL - A fatal alert message was received from our peer"` may appear.

2. The `Civetweb` web server running in the `WebServer` program displays the `index.html` page containing an authentication prompt.
3. The user clicks the `Log in` button.
4. The `WebServer` program queries the `AuthService` program via IPC to get the page containing the user name and password input form.
5. The `AuthService` program performs the following actions:
   - Generates a private key and public settings, and calculates the public key based on the Diffie-Hellman algorithm.
   - Creates the `auth.html` page containing the user name and password input form (the page code contains the public settings and the public key).
   - Transfers the received page to the `WebServer` program via IPC.
6. The `Civetweb` web server running in the `WebServer` program displays the `auth.html` page containing the user name and password input form.
7. The user completes the form and clicks the `Submit` button (correct data for authentication is contained in the file `secure_login/auth_service/src/authservice.cpp`).
8. The `auth.html` page code executed by the browser performs the following actions:
   - Generates a private key and calculates the public key and shared secret key based on the Diffie-Hellman algorithm.

- Encrypts the password by using the `XOR` operation with the shared secret key.
- Transmits the user name, encrypted password and public key to the web server.

9. The `WebServer` program queries the `AuthService` program via IPC to get the page containing the authentication result by transmitting the user name, encrypted password and public key.
10. The `AuthService` program performs the following actions:
    - Calculates the shared secret key based on the Diffie-Hellman algorithm.
    - Decrypts the password by using the shared secret key.
    - Returns the `result_err.html` page or `result_ok.html` page depending on the authentication result.
11. The `Civetweb` web server running in the `WebServer` program displays the `result_err.html` page or the `result_ok.html` page.

This way, confidential data is transmitted only in encrypted form through the network and web server. In addition, all HTTP traffic is transmitted through the network in encrypted form. Data is transferred between components via IPC interactions controlled by the Kaspersky Security Module.

Unit testing using the GoogleTest framework

In addition to the Information Obscurity pattern, the `Secure Login` example demonstrates use of the GoogleTest framework to conduct unit testing of applications developed for KasperskyOS (this framework is provided in KasperskyOS Community Edition).

The source code of the tests is located at the following path:

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

These unit tests are designed for verification of certain CPP modules of the authentication subsystem and web server.

To start testing:

1. Go to the directory with the `Secure Login` example.
2. Delete the `build` directory containing the results of the previous build by running the following command:

   ```
   sudo rm -rf build/
   ```

3. Run the command to start testing:

   ```
   $ RUN_TESTS=YES ./cross-build.sh
   ```

Tests are conducted in the `TestEntity` program. The `AuthService` and `WebServer` programs are not started in this case. Therefore, the example cannot be used to demonstrate the Information Obscurity pattern when testing is being conducted.

After testing is finished, the results of the tests are displayed.

Example files

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

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

Building and running example

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

```
$ cd build/einit
# Before running the following command, be sure that the path to
# the directory with the qemu-system-aarch64 executable file is saved in
# the PATH environment variable. If it is not there,
# add it to the PATH variable.
$ 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
```

See also Building and running examples section.

To ensure that the `secure_login` example will correctly run in Raspberry Pi, you must do the following after building the example and preparing your bootable SD card:

- Copy the `certs` and `www` directories located at the path `/opt/KasperskyOS-Community-Edition-<version>/examples/secure_login/resources/hdd` to the root directory of the bootable SD card.
- Create the `/lib` directory on the bootable SD card if this directory doesn't already exist.
- Open the `build/hdd/lib` directory that was generated when building the example and copy the directory contents to the `/lib` directory on the bootable SD card.

Developing drivers for KasperskyOS

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

To study the principles of developing drivers for KasperskyOS, the SDK includes the source code of some drivers that were developed using the `kdf` library (but they are only included in the DEB package for working with the Radxa Rock 3A hardware platform). The source code of drivers is located in the directory `/opt/KasperskyOS-Community-Edition-<version>/drivers`.

Appendices

This section provides additional information to supplement the primary text of the document.

Examples in KasperskyOS Community Edition

In addition to the Developer's Guide, the KasperskyOS Community Edition SDK provides examples of KasperskyOS-based solutions. The examples are located in the directory `/opt/KasperskyOS-Community-Edition-<version>/examples`. Each example resides in a separate directory containing the following:

- Directories containing the source code of programs.
- The `resources` directory contains the formal specification of the solution component. This directory may optionally contain files necessary for the example to work. For example, it may contain the network configuration files `hosts`, `dhcpcd.conf` and `ntp.conf`.
- Solution build scenarios in the form of `CMakeLists.txt` files.
- `README.md` files. These files have the same type of contents and refer to this Guide.
- [Optional] The `vendor` directory contains the libraries and their metadata required for managing dependencies in Rust-based projects.

> Before running the examples on Radxa ROCK 3A, you also need to build the drivers provided as a source code in the SDK. Instructions for building drivers can be found in the descriptions of examples (README.md files).

The table below provides a description of examples for developing basic solutions.

Basic solutions

Example directory in the SDK	Brief description of the example	Running in QEMU	Running in Raspberry Pi 4 B	Running on Radxa ROCK 3A
hello	Demonstrates the most basic solution. The `Hello` program sends a message to standard error. For more details, refer to hello example.	Yes	Yes	Yes
echo	Demonstrates interaction between programs via IPC. The `Client` sends data to the `Server` program, receives the modified data in return, and prints that data to standard error. For more details, refer to echo example.	Yes	Yes	Yes
ping	Demonstrates use of a solution security policy to control IPC interaction between programs. The two interface methods `Ping` and `Pong` provided by the `Server` program are called by the `Client` program in a varying sequence. However, the solution security policy allows only alternating calls of `Ping` and `Pong`. For more details, refer to ping example.	Yes	Yes	Yes
hello_from_rust	Demonstrates how to let your solution include a simple program that was developed based on Rust and built using the Cargo build system and package manager. The `Hello` program sends a message to standard error. For more details, refer to hello_from_rust example.	Yes	Yes	Yes
hello_corrosion	Demonstrates how to let your solution include a simple program that was developed based on Rust and uses Corrosion, which is a set of libraries for integrating Rust into CMake projects. The `Hello` program sends a message to standard error. For more details, refer to hello_corrosion example.	Yes	Yes	Yes

The table below provides a description of examples that demonstrate the use of file systems and a network stack.

Using file systems and a network stack in solutions

Example directory in the SDK	Brief description of the example	Running in QEMU	Running in Raspberry Pi 4 B	Running on Radxa ROCK 3A
embedded_vfs	Demonstrates how to include a system program that implements file systems and a network stack in a solution. The `Client` program creates a directory, creates and deletes a file, and reads from a file or writes to a file. For more details, refer to embedded_vfs example.	Yes	Yes	Yes
net_with_separate_vfs	Demonstrates how to establish a connection between programs running in KasperskyOS via TCP sockets using a loopback interface. The `Client` program creates a TCP socket, connects to the server through the loopback interface, sends data to the server, and closes the connection. The `Server` program accepts the connection request through the TCP socket, receives data from the client, and then closes the connection. For more details, refer to net_with_separate_vfs example.	Yes	Yes	Yes
net2_with_separate_vfs	Demonstrates how to establish a connection via TCP sockets between the client program running in KasperskyOS and the server program running in the host operating system. The `Client` program creates a TCP socket, connects to the server, sends data to the server, and closes the connection. The `Server` program accepts the connection request through the TCP socket, receives data from the client, and then closes the connection. For more details, refer to net2_with_separate_vfs example.	Yes	Yes	Yes
vfs_extfs	Demonstrates how to mount block device file systems (ext2, ext3, ext4). The `FileVfs` program implements the server portion of the virtual file system, thereby providing the interface used for interaction with file systems and block devices. The `Client` program sends requests to the `FileVfs` program via IPC to mount file systems to the specified directories and perform basic operations with files (create, write, read, and delete) to verify that the file systems are mounted correctly. For more details, refer to vfs_extfs example.	Yes	Yes	Yes
multi_vfs_ntpd	Demonstrates support of the Network Time Protocol (NTP). The `Ntpd` program is provided in KasperskyOS Community Edition and is an implementation of an NTP client, which synchronizes the system time with NTP servers. The `Client` program changes the current system time to the time specified in the macro. After the time is synchronized by the `Ntpd` program, the year in the received system time is expected to differ from the year that was previously set by the macro. For more details, refer to multi_vfs_ntpd example.	Yes	Yes	Yes
multi_vfs_dns_client	Demonstrates use of the Domain Name System (DNS) for domain name resolution in KasperskyOS. After checking the network connection, the `Client` program receives information about the DNS server by using its fully qualified domain name and port. Then it resolves the domain name to an IP address and verifies that the IP address matches the previously predefined values. For more details, refer to multi_vfs_dns_client example.	Yes	Yes	Yes
multi_vfs_dhcpcd	Demonstrates support of the network protocol named Dynamic Host Configuration Protocol (DHCP) in KasperskyOS. The `Dhcpcd` program is provided in KasperskyOS Community Edition and is an implementation of a DHCP client, which configures network interfaces. The `Client` program receives information about the configured network interfaces and prints them to standard error. For more details, refer to multi_vfs_dhcpcd example.	Yes	Yes	Yes
mqtt_publisher	Demonstrates support of the Message Queue Telemetry Transport (MQTT) protocol for exchanging messages in KasperskyOS. The `Publisher` program is an implementation of an MQTT publisher running in KasperskyOS that publishes the current time in a loop. The MQTT subscriber running in the host operating system prints messages received from the MQTT publisher to standard output. For more details, refer to mqtt_publisher example.	Yes	Yes	Yes
mqtt_subscriber	Demonstrates support of the Message Queue Telemetry Transport (MQTT) protocol for exchanging messages in KasperskyOS. The `Subscriber` program is an implementation of an MQTT subscriber running in KasperskyOS that prints messages to standard output when it receives those messages from the MQTT publisher running in the host operating system. For more details, refer to mqtt_subscriber example.	Yes	Yes	Yes

The table below provides a description of examples using the drivers provided in KasperskyOS Community Edition to work with GPIO, I2C, UART, SPI and USB hardware interfaces.

Example directory in the SDK	Brief description of the example	Running in QEMU	Running in Raspberry Pi 4 B	Running on Radxa ROCK 3A
gpio_input	Demonstrates use of the General-Purpose Input/Output (GPIO) driver for input via GPIO pins. The `Client` program configures the GPIO pins to output mode and reads their values. For more details, refer to gpio_input example.	No	Yes	Yes
gpio_output	Demonstrates use of the GPIO driver for output via GPIO pins. The `Client` program configures GPIO pins to output mode and switches them to a logical one (voltage on the pin) and then to a logical zero (no voltage on the pin). For more details, refer to gpio_output example.	No	Yes	Yes
gpio_interrupt	Demonstrates use of the GPIO driver to verify the operation of interrupts for GPIO pins. The `Client` program configures GPIO pins to input mode and registers interrupt handling functions for all GPIO pins except those specified in the list of exclusions. If an interrupt is triggered on a GPIO pin, the `Client` program removes the handler function for this GPIO pin. The program terminates when an interrupt occurs on each GPIO pin. For more details, refer to gpio_interrupt example.	No	Yes	Yes
gpio_echo	Demonstrates use of the GPIO driver to verify the input/output functionality of GPIO pins and the operation of interrupts for GPIO pins. The `Client` program configures two interconnected pins: one pin to input mode and one pin to output mode. It then registers the interrupt handler for the input pin and changes the state of the output pin several times while waiting for the interrupt handler to be triggered. For more details, refer to gpio_echo example.	No	Yes	Yes
i2c_ds1307_rtc	Demonstrates use of the Inter-Integrated Circuit (I2C) driver on the Raspberry Pi 4 B hardware platform. The `I2cClient` program verifies operation of the real-time clock module on the DS1307Z chip via the I2C driver interface by setting and verifying the date on the device. For more details, refer to i2c_ds1307_rtc example.	No	Yes	No
i2c_bm8563_rtc	Demonstrates use of the Inter-Integrated Circuit (I2C) driver on the Radxa ROCK 3A hardware platform. The `Bm8653` program verifies operation of the real-time clock on the BM8563 chip via the I2C driver interface by setting and verifying the date on the device. For more details, refer to i2c_bm8563_rtc example.	No	No	Yes
uart	Demonstrates use of the Universal Asynchronous Receiver-Transmitter (UART) driver. The `Client` program performs the following operations with UART: initialization, opening a port, sending data, and closing a port. For more details, refer to uart example.	Yes	Yes	Yes
spi_check_regs	Demonstrates use of the Serial Peripheral Interface (SPI) driver. The `Client` program opens an SPI channel, prints its parameters to standard output and sets the required operating mode. The program then uses this SPI channel to forward the data sequence and waits to receive the ID of the ATTiny controller installed on the Sense HAT board. For more details, refer to spi_check_regs example.	No	Yes	Yes
barcode_scanner	Demonstrates use of a Universal Serial Bus (USB) driver via the `libevdev` library. The `BarcodeScanner` program uses the `libevdev` library for interaction with a barcode scanner connected to the USB port of Raspberry Pi 4 B. For more details, refer to barcode_scanner example.	No	Yes	Yes
watchdog_system_reset	Demonstrates use of the Watchdog driver for monitoring the state of KasperskyOS and automatically restarting the Raspberry Pi 4 B. The `Client` program interacts with the Watchdog timer as follows: changes the default value of the timer to a new value and starts the timer, resets the timer several times, and waits for the operating system to restart when the timer is triggered. For more details, refer to watchdog_system_reset example.	No	Yes	No
mass_storage	Demonstrates use of the UsbMassStorage driver for working with an external USB drive connected to the USB port of the Raspberry Pi 4 B. The `FileVfs` program implements the server portion of the virtual file system, thereby providing the interface used for interaction with file systems and block devices. The `Client` program sends requests to the `FileVfs` program via IPC to mount file systems to the specified directories and perform basic operations with files (create, write, read, and delete) to verify that the file systems are mounted correctly on an external USB drive. For more details, refer to mass_storage example.	No	Yes	Yes
can_loopback	Demonstrates use of the CAN driver on the Radxa ROCK 3A hardware platform when there is no need to connect an additional peripheral device for CAN	No	No	Yes

	transceivers. In the example, the CAN port is configured and then a CAN packet is sent. The received CAN packet is checked to make sure that it matches the sent CAN packet. For more details, refer to can_loopback example.			
can2can	Demonstrates use of the CAN driver on the Radxa ROCK 3A hardware platform to forward a message between two CAN interfaces. In contrast to the can_loopback example, which uses only one CAN interface and does not require additional equipment, this example requires two CAN transceivers that are connected to the `CAN_H` and `CAN_L` lines. For more details, refer to can2can example.	No	No	Yes
adc_hello	Demonstrates use of the Sensors driver to check the ADC functionality on the Radxa ROCK 3A hardware platform. The `AdcHello` program calculates the voltage supplied to the ADC channel, normalizes it and prints it to standard output. For more details, refer to adc_hello example.	No	No	Yes

The table below provides examples of using the libraries provided in KasperskyOS Community Edition to perform various functions, such as logging, applying regular expressions, exchanging messages via IPC, and testing performance.

Using libraries in solutions

Example directory in the SDK	Brief description of the example	Running in QEMU	Running in Raspberry Pi 4 B	Running on Radxa ROCK 3A
shared_libs	Demonstrates the use of dynamic and static libraries in KasperskyOS. The `Client` program calls functions from a static library, a dynamic library linked together with the program, and a dynamic library loaded into memory when the `dlopen()` function of the POSIX interface is called. For more details, refer to shared_libs example.	Yes	Yes	Yes
koslogger	Demonstrates use of the `spdlog` logging library in KasperskyOS via the `KOSLogger` wrapper library. The `Client` program writes messages to a log, which is saved on an SD card when an example is started on the Raspberry Pi 4 B or in the `sdcard0.img` image file when an example is run in QEMU. Messages are written using various log levels. For more details, refer to koslogger example.	Yes	Yes	Yes
pcre	Demonstrates use of the `pcre` library for working with regular expressions in KasperskyOS. The `Client` program searches for all occurrences of the necessary word within the defined text fragment and counts the number of these occurrences. For more details, refer to pcre example.	Yes	Yes	Yes
iperf_separate_vfs	Demonstrates use of the `iperf` library to test network performance. The `Server` program configures the network interface, then creates and runs a network performance test using the `iperf` library. For more details, refer to iperf_separate_vfs example.	Yes	Yes	Yes
messagebus	Demonstrates use of the `MessageBus` component in KasperskyOS. In this example, the `Publisher`, `SubscriberA` and `SubscriberB` programs use the `MessageBus` component to exchange messages via IPC. For more details, refer to messagebus example.	Yes	Yes	Yes
pal_tests	Demonstrates use of the Policy Assertion Language (PAL) when writing tests for a KasperskyOS-based solution security policy. These tests check the solution security policy prior to starting development of the software code based on the formal specifications of solution components. For more details, refer to pal_tests example.	Yes	Yes	Yes

The table below shows examples of using security patterns in KasperskyOS.

Using security patterns in solutions

Example directory in the SDK	Brief description of the example	Running in QEMU	Running in Raspberry Pi 4 B	Running on Radxa ROCK 3A
secure_logger	Demonstrates use of the Distrustful Decomposition pattern for separating event log read/write operations. The `Application` program generates a message and forwards it to the `Logger` program via IPC. The `Logger` program receives the message and writes it to the event log residing on the drive. The `Reader` program reads the message from the event log and forwards it to the `LogViewer` program via IPC. The `LogViewer` program prints the message to standard output. For more details, refer to secure_logger example.	Yes	Yes	Yes

separate_storage	Demonstrates use of the Distrustful Decomposition pattern to separate data storage for trusted and untrusted programs. The example contains two programs: `UserManager` and `CertificateManager`. Each of these programs uses its own instance of the VFS program to access a separate file system. Each VFS program instance includes a block device driver linked to an individual logical drive partition. The `UserManager` program does not have access to the file system of the `CertificateManager` program, and vice versa. For more details, refer to separate_storage example.	Yes	Yes	Yes
defer_to_kernel	Demonstrates use of the Defer to Kernel and Policy Decision Point patterns to guarantee isolation of running programs (processes) by the KasperskyOS kernel. In this example, the `ValidPictureClient` and `NonValidPictureClient` programs query the `PictureManager` program to receive information. However, only the `ValidPictureClient` program is allowed to interact with the `PictureManager` program. For more details, refer to defer_to_kernel example.	Yes	Yes	Yes
device_access	Demonstrates use of the Privilege Separation pattern in which different programs are responsible for authorization and access to data. The `Device` program queries the `Storage` program to receive information and queries the `LoginManager` program for authorization. The `Device` program can query the `Storage` program only after successful authorization in the `LoginManager` program. For more details, refer to device_access example.	Yes	Yes	Yes
secure_login	Demonstrates use of the Information Obscurity pattern for the capability to transmit critical system information through an untrusted environment. The `Civetweb` web server interacts with the browser to process authentication requests, and data is transmitted between the browser and the server via the TLS terminator for traffic encryption. The `AuthService` program generates encryption keys to encrypt passwords based on the Diffie-Hellman algorithm. For more details, refer to secure_login example.	Yes	Yes	Yes

hello example

The `hello.c` code looks familiar and simple to a developer that uses C, and is fully compatible with POSIX:

hello.c

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

int main(int argc, const char *argv[])

{

    fprintf(stderr,"Hello world!\n");

    return EXIT_SUCCESS;

}
```

Compile this code using `aarch64-kos-clang`, which is included in the development tools of KasperskyOS Community Edition:

```
aarch64-kos-clang -o Hello hello.c
```

> The program name (and, consequently, the name of the executable file) must begin with an uppercase letter.

EDL description of the Hello process class

A static description of the `Hello` program consists of a single file named `Hello.edl` that must indicate the name of the process class:

```
Hello.edl

/* The process class name follows the reserved word "entity". */
entity Hello
```

> The process class name must begin with an uppercase letter. The name of an EDL file must match the name of the class that it describes.

Creating the Einit initializing program

> When KasperskyOS is loaded, the kernel starts a program named `Einit`. The `Einit` program starts all other programs included in the solution, which means that it serves as the *initializing program*.
> The KasperskyOS Community Edition toolkit includes the einit tool, which generates the code of the initializing program (`einit.c`) based on the *init description*. In the example provided below, the file containing the init description is named `init.yaml`, but it can have any name.
> For more details, refer to "Starting processes".

If you want the `Hello` program to start after the operating system is loaded, all you need to do is specify its name in the `init.yaml` file and build an `Einit` program based on it.

```
init.yaml

entities:
# Start the "Hello" program.
- name: Hello
```

Building the security module

The hello example contains a basic solution security policy (`security.psl`) that allows all interactions.

The security module (`ksm.module`) is built based on `security.psl`.

Example files

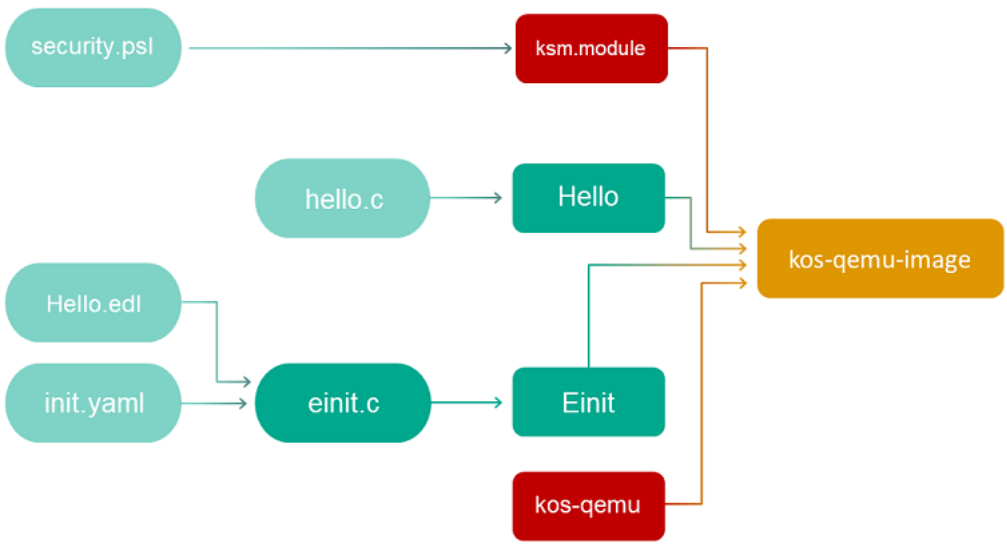
The code of the example and build scripts are available at the following path:

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

Building and running example

See Building and running examples section.

The general build scheme for the hello example looks as follows:



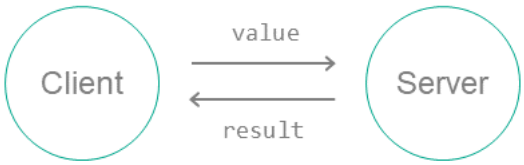
echo example

The echo example demonstrates the use of IPC transport.

It shows how to use the main tools that let you implement interaction between programs.

The echo example describes a basic case of interaction between two programs:

1. The `Client` program sends a number (`value`) to the `Server` program.
2. The `Server` program modifies this number and sends the new number (`result`) to the `Client` program.
3. The `Client` program prints the `result` number to the screen.

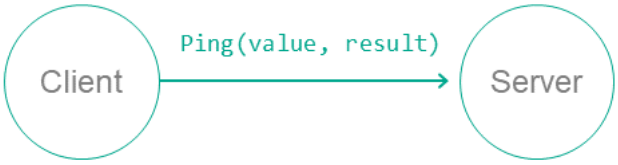


To set up this interaction between programs:

1. Connect the `Client` and `Server` programs by using the init description.
2. On the server, implement an interface with a single `Ping` method that has one input argument (the original number (`value`)) and one output argument (the modified number (`result`)).

   Description of the `Ping` method in the IDL language:

   ```
   Ping(in UInt32 value, out UInt32 result);
   ```



3. Create static description files in the EDL, CDL and IDL languages. Use the NK compiler to generate files containing transport methods and types (proxy object, dispatchers, etc.).

4. In the code of the `Client` program, initialize all required objects (transport, proxy object, request structure, etc.) and call the interface method.

5. In the code of the `Server` program, prepare all the required objects (transport, component dispatcher and program dispatcher, etc.), accept the request from the client, process it and send a response.

Example files

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

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

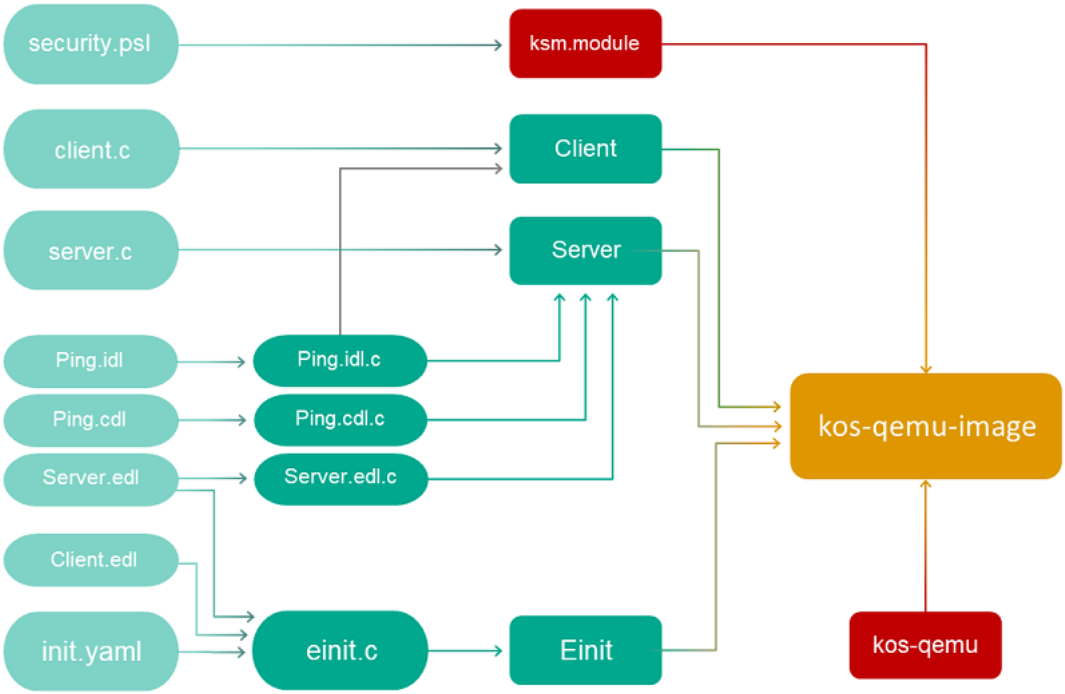
The echo example consists of the following source files:

- `client/src/client.c` contains implementation of the `Client` program.
- `server/src/server.c` contains implementation of the `Server` program.
- `resources/Server.edl`, `resources/Client.edl`, `resources/Responder.cdl`, `resources/Pingable.idl` are static descriptions.
- `init.yaml` contains the init description.

Building and running example

See the Building and running examples section.

The build scheme for the echo example looks as follows:



ping example

The ping example demonstrates the use of a solution security policy to control interactions between programs.

The ping example includes four programs: `Client`, `Server`, `KlogEntity` and `KlogStorageEntity`.

The `Server` program provides two identical `Ping` and `Pong` methods that receive a number and return a modified number:

```
Ping(in UInt32 value, out UInt32 result);
Pong(in UInt32 value, out UInt32 result);
```

The `Client` program calls both of these methods in a different sequence. If the method call is denied by the solution security policy, a message regarding the failed call attempt is displayed.

The system programs `KlogEntity` and `KlogStorageEntity` perform a security audit.

> The transport part of the ping example is virtually identical to its counterpart in the echo example. The only difference is that the ping example uses two methods (`Ping` and `Pong`) instead of just one.

Solution security policy in the ping example

The solution security policy in this example allows startup of the KasperskyOS kernel and the Einit program, which is allowed to start all programs in the solution. Queries to the `Server` program are managed by methods of the Flow security model.

The finite-state machine described in the configuration of the `request_state` Flow security model object has two states: `not_sent` and `sent`. The initial state is `not_sent`. Only transitions from `not_sent` to `sent` and vice versa are allowed.

When the `Ping` and `Pong` methods are called, the current state of the `request_state` object is checked. In the `not_sent` state, only a `Ping` call is allowed, in which case the state changes to `sent`. Likewise, in the `sent` state, only a `Pong` call is allowed, in which case the state changes to `not_sent`.

Therefore, the `Ping` and `Pong` methods can be called only in succession.

Fragment of the security.psl file

```
/* Solution security policy for demonstrating use of the
 * Flow security model in the ping example */

/* Include PSL files containing formal representations of
 * Base and Flow security models */
use nk.base._
use nk.flow._

/* Including EDL files */
use EDL Einit
use EDL ping.Client
use EDL ping.Server

/* Create Flow security model object */
policy object request_state : Flow {
    type States = "not_sent" | "sent"
    config = {
        states      : [ "not_sent", "sent" ],
        initial     : "not_sent",
        transitions : {
            "not_sent" : [ "sent" ],
            "sent"     : [ "not_sent" ]
        }
    }
}

/* When the Einit program starts the Server program,
 * the initial state is set for the finite-state machine */
execute src=Einit dst=ping.Server method=main {
    request_state.init { sid: dst_sid }
}

/* When a client of the ping.Client class calls the Ping method of the controlimpl.connectionimpl endpoint
 * of a server of the ping.Server class, the system checks whether the request_state object is
 * in the "not_sent" state. If it is, receipt of the request is allowed and
 * the request_state object is set to the "sent" state. */
request src=ping.Client dst=ping.Server endpoint=controlimpl.connectionimpl method=Ping {
    request_state.allow { sid: dst_sid, states: [ "not_sent" ] }
    request_state.enter { sid: dst_sid, state: "sent" }
}

/* When a client of the ping.Client class calls the Pong method of the controlimpl.connectionimpl endpoint
 * of a server of the ping.Server class, the system checks whether the request_state object is
 * in the "sent" state. If it is, receipt of the request is allowed and
 * the request_state object is set to the "not_sent" state. */
request src=ping.Client dst=ping.Server endpoint=controlimpl.connectionimpl method=Pong {
    request_state.allow { sid: dst_sid, states: [ "sent" ] }
    request_state.enter { sid: dst_sid, state: "not_sent" }
}
/* A server of the ping.Server class is allowed to respond to queries from a client of the ping.Client class
 * that calls the Ping and Pong methods of the controlimpl.connectionimpl endpoint. */
response src=ping.Server dst=ping.Client endpoint=controlimpl.connectionimpl {
    match method=Ping { grant () }
    match method=Pong { grant () }
}
```

The security policy description in the ping example also contains a section for solution security policy tests.

For an example of such a policy, see the "Example 2" section in "Examples of tests for KasperskyOS-based solution security policies".

The full security policy description for the ping example is located in the `security.psl.in` and `core.psl` files at the following path: `/opt/KasperskyOS-Community-Edition-<version>/examples/ping/einit/src`.

Example files

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

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

Building and running the example

See Building and running examples section.

net_with_separate_vfs example

This example presents a basic case of network interaction using Berkeley sockets.

The example consists of `Client` and `Server` programs linked by a TCP socket using a loopback interface. Standard POSIX functions are used in the code of the programs.

To connect programs using a socket through a loopback, they must use the same network stack instance. This means that they must interact with a "shared" VFS program (in this example, this program is called `VfsNet`).

The CMake system, which is included with KasperskyOS Community Edition, is used to build and run the example.

Example files

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

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

Building and running example

See Building and running examples section.

net2_with_separate_vfs example

This example demonstrates the special features of a solution in which a program uses standard POSIX functions to interact with an external server.

The `net2_with_separate_vfs` example is a modified `net_with_separate_vfs` example. In contrast to the `net_with_separate_vfs` example, in this example a program interacts over the network with an external server rather than another program running in KasperskyOS.

This example consists of the `Client` program running in KasperskyOS on QEMU or Raspberry Pi and the `Server` program running in a Linux host operating system. The `Client` program and `Server` program are bound by a TCP socket. Standard POSIX functions are used in the code of the `Client` program.

To connect the `Client` program and the `Server` program using a socket, the `Client` program must interact with the `VfsNet` program. During the build, the `VfsNet` program is linked to a network driver that supports interaction with the `Server` program running in Linux.

Example files

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

```
/opt/KasperskyOS-Community-Edition-<version>/examples/net2_with_separate_vfs
```

The IP address of the `Server` program is defined in the `./CMakeList.txt` file by using the `SERVER_IP` variable and is set to `10.0.2.2` by default. The default values for the `Client` program (interface name, address, network mask, and gateway address) are taken from the file `/opt/KasperskyOS-Community-Edition-<version>/sysroot-aarch64-kos/include/kos_net.h`. You can change these values according to the configuration of your network in the `client/src/client.c` file located in the example directory.

Building and running the example

The CMake system, which is included with KasperskyOS Community Edition, is used to build and run the example. See Building and running examples section.

To ensure that an example runs correctly, you must run the `Server` program in a Linux host operating system or on a computer connected to Raspberry Pi.

If the example is run in QEMU, the `Server` program will be built and run in the Linux host operating system automatically by the `cross-build.sh` script. After performing the build, the `server` executable file of the `Server` program is located in the following directory:

```
/opt/KasperskyOS-Community-Edition-<version>/examples/net2_with_separate_vfs/build/host/server/
```

If the example is run on Raspberry Pi, you must independently build the executable file of the `Server` program by running the following commands:

```
$ cd net2_with_separate_vfs/server/src/
$ gcc -o server server.c
```

embedded_vfs example

This example demonstrates how to embed the virtual file system (VFS) provided in KasperskyOS Community Edition into a program being developed.

In this example, the `Client` program fully encapsulates the VFS implementation from KasperskyOS Community Edition. This lets you eliminate the use of IPC for all the standard I/O functions (`stdio.h`, `socket.h`, etc.) for debugging or performance improvement purposes, for example.

The `Client` program tests the following operations:

- Create a folder.

- Create and delete a file.
- Read from a file and write to a file.

Supplied resources

For QEMU, the example includes the `hdd.img` image of a hard drive with the FAT32 file system.

Example files

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

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

Building and running example

See Building and running examples section.

vfs_extfs example

This example demonstrates how to mount file systems of a block device.

The `FileVfs` program implements the server portion of the virtual file system, thereby providing the interface used for interaction with file systems and block devices. The `Client` program sends requests to the `FileVfs` program via IPC to mount file systems (`ext2`, `ext3`, `ext4`) to the specified directories and perform basic operations with files (create, write, read, and delete) to verify that the file systems are mounted correctly.

Example files

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

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

Building and running example

See Building and running examples section.

Preparing an SD card to run on Raspberry Pi 4 B

To run the `vfs_extfs` example on Raspberry Pi 4 B, the SD card must have a bootable partition with the solution image as well as 3 additional partitions with the `ext2`, `ext3` and `ext4` file systems, respectively.

multi_vfs_ntpd example

This example demonstrates support of the Network Time Protocol (NTP) for synchronizing the system time in KasperskyOS.

Example directory in the SDK

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

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

List of programs

- `Client` – application that changes the current system time.
- `Ntpd` – system program that implements the NTP client that synchronizes the system time with NTP servers.
- `Dhcpcd` – system program that implements the DHCP client.
- `VfsSdCardFs` – system program that supports the file system of SD cards.
- `VfsNet` – system program that supports network protocols.
- `EntropyEntity` – system program that implements random number generation.
- `DNetSrv` – network card driver.
- `SDCard` – SD card driver.
- `BSP` – driver for configuring pin multiplexing settings (pinmux).
- `Bcm2711MboxArmToVc` – driver for working with the VideoCore (VC6) coprocessor via mailbox technology for Raspberry Pi 4 B.

Initialization description

The solution initialization description file named `init.yaml` is generated during the solution build process based on the following template:

`./einit/src/init.yaml.in`

When building the example, the `@INIT_Client_ENTITY_CONNECTIONS+@` macro in the `init.yaml.in` template in the `init.yaml` file is replaced by a list of IPC channels with all the system programs that are linked to the `Client program.` This list serves as a supplement to the manually defined IPC channels in the `init.yaml.in` template.

When building the example, the `@INIT_FileVfs_ENTITY_CONNECTIONS@` macro in the `init.yaml` file is replaced by a list of IPC channels with all the system programs that are linked to the `FileVfs` program.

During the build, the `@INIT_EXTERNAL_ENTITIES@` macro in the `init.yaml.in` template in the `init.yaml` file is replaced by a list of system programs that are linked to applications. This list contains the IPC channels of system programs, the run parameters of the `main()` function, and the values of environment variables.

For more details, see init.yaml.in template.

Security policy description

The `security.psl` file contains the solution security policy description and is generated based on the following template during the solution build process:

`./einit/src/security.psl.in`

Resources

- The `./resources/edl` directory contains the `Client.edl` file, which provides the formal specification of the KasperskyOS-based solution component.
- The directory `./resources/hdd/etc` contains the configuration files for the `VfsNet`, `Dhcpcd` and `Ntpd` programs: `hosts`, `dhcpcd.conf` and `ntp.conf`, respectively.

Operating scenario

The `Client` program changes the current system time to the time specified in the macro. After the time is synchronized by the `Ntpd` program, the year in the received system time is expected to differ from the year that was previously set by the macro.

Building and running the example

See Building and running examples section.

multi_vfs_dns_client example

This example shows how to use an external DNS server in KasperskyOS.

The example also demonstrates the use of various virtual file systems (VFS) in a single solution:

- The `VfsNet` program is used for working with the network.
- The `VfsSdCardFs` program is used to work with the file system.

The `Client` program uses standard `libc` library functions for contacting an external DNS service. These functions are converted into queries to the `VfsNet` program via IPC.

The CMake system, which is included with KasperskyOS Community Edition, is used to build and run the example.

Supplied resources

- The directory `./resources/edl` contains the `Client.edl` file, which contains a static description of the `Client` program.

- The directory `./resources/hdd/etc` contains the configuration files for the `VfsNet` and `Dhcpcd` programs: `hosts` and `dhcpcd.conf`, respectively.

Example files

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

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

Building and running example

See Building and running examples section.

multi_vfs_dhcpcd example

Example use of the `kl.rump.Dhcpcd` program.

The `Dhcpcd` program is an implementation of a DHCP client, which gets network interface parameters from an external DHCP server in the background and passes them to a virtual file system (hereinafter referred to as a VFS).

The example also demonstrates the use of different VFSes in a single solution. The example uses different VFS to access the functions for working with the file system and functions for working with the network:

- The `VfsNet` program is used for working with the network.

- The `VfsSdCardFs` program is used to work with the file system.

The `Client` program uses standard `libc` library functions for getting information on network interfaces (`ioctl`). These functions are converted into queries to the VFS program via IPC.

The CMake system, which is included with KasperskyOS Community Edition, is used to build and run the example.

Supplied resources

The `./resources/hdd/etc` directory contains configuration files for the VFS and `Dhcpcd` programs. The standard syntax of `dhcpcd.conf` is used for the `Dhcpcd` program configuration.

The `CMakeLists.txt` root file defines the values of variables that determine the selected configuration file:

- `DHCPCD_FALLBACK`

  Dynamically receive the parameters of network interfaces from an external DHCP server but statically define the parameters if the DHCP server is not available. This value is used by default.

- `DHCPCD_DYNAMIC`

  Dynamically receive the parameters of network interfaces from an external DHCP server.

- `DHCPCD_STATIC`

  Statically define the parameters of network interfaces.

Example files

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

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

Building and running example

See Building and running examples section.

mqtt_publisher (Mosquitto) example

Example use of the MQTT protocol in KasperskyOS.

In this example, an MQTT subscriber must be started on the host operating system, and an MQTT publisher must be started on KasperskyOS. The `Publisher` program is an implementation of an MQTT publisher that publishes the current time with a 5-second interval.

When the example starts and runs successfully, an MQTT subscriber started on the host operating system prints a `"received PUBLISH"` message with a `"datetime"` topic.

The example also demonstrates the use of various virtual file systems (VFS) in a single solution:

- The `VfsNet` program is used for working with the network.
- The `VfsSdCardFs` program is used to work with the file system.

The CMake system, which is included with KasperskyOS Community Edition, is used to build and run the example.

Starting Mosquitto

To run this example, a Mosquitto MQTT broker must be installed and started on the host system. To install and start Mosquitto, run the following commands:

```
$ sudo apt install mosquitto mosquitto-clients
$ sudo /etc/init.d/mosquitto start
```

To start an MQTT subscriber on the host system, run the following command:

```
$ mosquitto_sub -d -t "datetime"
```

Supplied resources

- The directory `./resources/edl` contains the `Publisher.edl` file, which contains a static description of the `Publisher` program.
- The directory `./resources/hdd/etc` contains the configuration files for the `VfsNet`, `Dhcpcd` and `Ntpd` programs: `hosts`, `dhcpcd.conf` and `ntp.conf`, respectively.

Example files

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

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

Building and running example

See Building and running examples section.

mqtt_subscriber (Mosquitto) example

Example use of the MQTT protocol in KasperskyOS.

In this example, an MQTT publisher must be started on the host operating system, and an MQTT subscriber must be started on KasperskyOS. The `Subscriber` program is an implementation of an MQTT subscriber.

When the example starts and runs successfully, an MQTT subscriber started on KasperskyOS prints a `"Got message with topic: my/awesome/topic, payload: hello"` message.

The example also demonstrates the use of various virtual file systems (VFS) in a single solution:

- The `VfsNet` program is used for working with the network.
- The `VfsSdCardFs` program is used to work with the file system.

The CMake system, which is included with KasperskyOS Community Edition, is used to build and run the example.

Starting Mosquitto

To run this example, a Mosquitto MQTT broker must be installed and started on the host system. To install and start Mosquitto, run the following commands:

```
$ sudo apt install mosquitto mosquitto-clients
$ sudo /etc/init.d/mosquitto start
```

To start an MQTT publisher on the host system, run the following command:

```
$ mosquitto_pub  -t "my/awesome/topic" -m "hello"
```

Supplied resources

- The directory `./resources/edl` contains the `Subscriber.edl` file, which contains a static description of the `Subscriber` program.
- The directory `./resources/hdd/etc` contains the configuration files for the `VfsNet`, `Dhcpcd` and `Ntpd` programs: `hosts`, `dhcpcd.conf` and `ntp.conf`, respectively.

Example files

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

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

Building and running example

See Building and running examples section.

gpio_input example

Example use of the GPIO driver.

This example lets you verify the functionality of GPIO input pins. The total number of pins on the card is defined by the `GPIO_PIN_NUM` macro. All pins except those indicated in `ExceptionPinArr` array are set for input by default. The voltage on the pins corresponds to the state of the registers of the pull-up resistors. The state of all pins will be read in succession. Messages about the state of the pins will be displayed on the console. The delay between the readings of adjacent pins is determined by the `DELAY_S` macro (the time is indicated in seconds).

`ExceptionPinArr` is an array of GPIO pin numbers that need to be excluded from the example. This may be necessary if some pins are already being used for other functions, e.g. if pins are being used for a UART connection during debugging.

If you build and run this example on QEMU, an error will occur. This is the expected behavior, because there is no GPIO driver for QEMU.

If you build and run this example for Raspberry Pi 4 or Radxa ROCK 3a, an error will occur.

Example files

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

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

Building and running example

See Building and running examples section.

gpio_output example

Example use of the GPIO driver.

This example lets you verify the functionality of GPIO output pins. The total number of pins on the card is defined by the `GPIO_PIN_NUM` macro. All pins other than those indicated in the `ExceptionPinArr` array are configured for output. Each pin will be sequentially changed to a logical one (voltage on the pin) and then to a logical zero. The delay between the changes of pin state is defined by the `DELAY_S` macro (the time is indicated in seconds).

`ExceptionPinArr` is an array of GPIO pin numbers that need to be excluded from the example. This may be necessary if some pins are already being used for other functions, e.g. if pins are being used for a UART connection during debugging.

If you build and run this example on QEMU, an error will occur. This is the expected behavior, because there is no GPIO driver for QEMU.

If you build and run this example for Raspberry Pi 4 or Radxa ROCK 3a, an error will occur.

Example files

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

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

Building and running example

See Building and running examples section.

gpio_interrupt example

Example use of the GPIO driver.

This example lets you verify the functionality of GPIO pin interrupts. The total number of pins on the card is defined by the `GPIO_PIN_NUM` macro. In the `pinsBitmap` bitmask, pins from the `ExceptionPinArr` array are marked as handled so that the example can properly terminate later. All pins other than those indicated in the `ExceptionPinArr` array are switched to the `PINS_MODE` state. An interrupt handler function will be registered for all pins other than those indicated in the `ExceptionPinArr` array.

In an endless loop, the example checks whether the `pinsBitmap` bitmask is equal to the `DONE_BITMASK` bitmask (which corresponds to the condition when an interrupt has occurred on each GPIO pin). The interrupt handler function is then called in this loop. If an interrupt occurs for the first time on a pin that is not specified in the `ExceptionPinArr` array, the interrupt handler function uses the `GpioGetEvent` function to receive this event. The corresponding pin in the `pinsBitmap` bitmask is marked as handled. Interrupt detection on this pin is disabled by calling the `GpioReleaseMode` function.

Keep in mind how the example may be affected by the initial state of the registers of pull-up resistors for each pin.

Some GPIO pins may also include external pull-up resistors (for example, they are on pins GPIO2 and GPIO3 in revisions 1.4 and 1.5 of the Raspberry Pi 4 B platform). If these types of resistors are present, weaker internal resistors will not allow these pins to be pulled to 0.

Interrupts for the `GPIO_EVENT_LOW_LEVEL` and `GPIO_EVENT_HIGH_LEVEL` events are not supported.

`ExceptionPinArr` is an array of GPIO pin numbers that need to be excluded from the example. This may be necessary if some pins are already being used for other functions, e.g. if pins are being used for a UART connection during debugging.

If you build and run this example on QEMU, an error will occur. This is the expected behavior, because there is no GPIO driver for QEMU.

If you build and run this example for Raspberry Pi 4 or Radxa ROCK 3a, an error will occur.

Example files

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

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

Building and running example

See Building and running examples section.

gpio_echo example

Example use of the GPIO driver.

This example makes it possible to verify the functionality of GPIO pins as well as the operation of GPIO interrupts. The port `gpio0` is used for the verification. The output pin (the pin number is defined in the `GPIO_PIN_OUT` macro) should be connected to the input pin (`GPIO_PIN_IN`). The output pin (`GPIO_PIN_OUT`) and the input pin (`GPIO_PIN_IN`) are configured. Use of the input pin is configured in the `IN_MODE` macro. The interrupt handler for the input pin is registered. The state of the output pin changes several times. If the example works correctly, then when the state of the output pin changes the interrupt handler will be called and will display the state of the input pin. What's more, the state of the output pin and the input pin must match.

If you build and run this example on QEMU, an error will occur. This is the expected behavior, because there is no GPIO driver for QEMU.

If you build and run this example for Raspberry Pi 4 or Radxa ROCK 3a, an error will occur.

Example files

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

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

Building and running example

See Building and running examples section.

koslogger example

This example demonstrates use of the `spdlog` library in KasperskyOS using the `KOSLogger` wrapper library.

In this example, the `Client` program creates log entries that are saved on an SD card (when running the example on Raspberry Pi) or in the image file named `build/einit/sdcard0.img` (when running the example in QEMU).

The example also demonstrates the use of various virtual file systems (VFS) in a single solution. The example uses different VFS to access the functions for working with the file system and functions for working with the network:

- The `VfsNet` program is used for working with the network.
- The `VfsSdCardFs` program is used to work with the file system.

The `kl.Ntpd` program is included in KasperskyOS Community Edition and is an implementation of an NTP client, which gets time parameters from external NTP servers in the background and passes them to the KasperskyOS kernel.

The `kl.rump.Dhcpcd` program is included in KasperskyOS Community Edition and is an implementation of a DHCP client, which gets the parameters of network interfaces from an external DHCP server in the background and passes them to the virtual file system.

The `CMake` system, which is included with KasperskyOS Community Edition, is used to build and run the example.

Example files

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

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

Building and running example

See Building and running examples section.

To ensure that the `koslogger` example will correctly run in Raspberry Pi, you must do the following after building the example and preparing your bootable SD card:

- Create the `/lib` directory on the bootable SD card if this directory doesn't already exist.
- Open the `build/hdd/lib` directory that was generated when building the example and copy the directory contents to the `/lib` directory on the bootable SD card.

pcre example

This example demonstrates use of the `pcre` library in KasperskyOS.

In this example, the `Client` program uses the `pcre` library and prints the results to the console.

The `CMake` system, which is included with KasperskyOS Community Edition, is used to build and run the example.

Example files

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

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

Building and running example

See Building and running examples section.

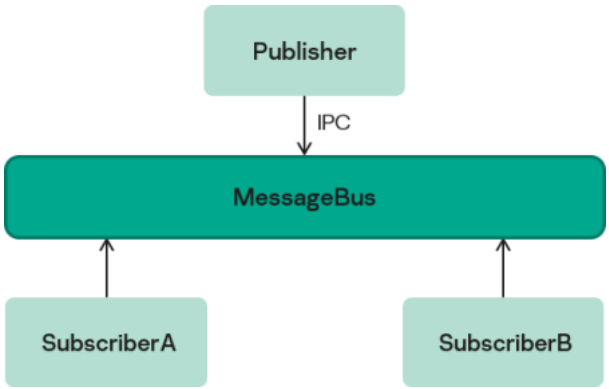
To ensure that the `pcre` example will correctly run in Raspberry Pi, you must do the following after building the example and preparing your bootable SD card:

- Create the `/lib` directory on the bootable SD card if this directory doesn't already exist.
- Open the `build/hdd/lib` directory that was generated when building the example and copy the directory contents to the `/lib` directory on the bootable SD card.

messagebus example

This example demonstrates use of the MessageBus component in KasperskyOS.

In this example, the `Publisher`, `SubscriberA` and `SubscriberB` programs use the `MessageBus` component to exchange messages:



Example of using the MessageBus component in a KasperskyOS-based solution

The `MessageBus` component implements the message bus that ensures receipt, distribution, and delivery of messages between programs. Use of a message bus in KasperskyOS-based solutions lets you easily scale the distribution of messages for a large number of subscribers, for example. The message bus can process large volumes of messages and effectively distribute them between recipients.

The log level of the `MessageBus` component is set by the `LOG_LEVEL` environment variable defined in the `./einit/CMakeLists.txt` file. This variable can have the following values:

- LOG_TRACE
- LOG_DEBUG
- LOG_INFO (default value)
- LOG_WARNING
- LOG_ERROR
- LOG_CRITICAL
- LOG_OFF

The `Publisher` program is the publisher that transfers messages to the bus:

1. The `IProviderFactory::CreateBusControl()` method is used to get the interface for registering a publisher in the message bus.
2. The `IProviderFactory::CreateBus()` method is used to get the interface containing the methods enabling the publisher to send messages to the bus.
3. The `IProviderControl::RegisterPublisher()` method is used to register the publisher in the message bus.
4. The `IProvider::Push()` method is used to send messages to the bus.
5. The `IProviderControl::UnregisterPublisher()` method is used to deregister a publisher in the message bus.

The `SubscriberA` and `SubscriberB` programs are the subscribers that receive messages from the bus:

1. The `IProviderFactory::CreateBusControl()` method is used to get the interface for registering a subscriber in the message bus.
2. The `IProviderFactory::CreateSubscriberRunner()` method is used to get the interfaces containing the methods enabling the subscriber to receive messages from the bus.
3. The `IProviderControl::RegisterSubscriber()` method is used to register the subscriber in the message bus.
4. The `ISubscriberRunner::Run()` method is used to switch a subscriber to standby mode to wait for a message from the bus.
5. The `ISubscriber::OnMessage()` method is called when a message is received from the bus.
6. The `IProviderControl::UnregisterSubscriber()` method is used to deregister a subscriber in the message bus.

The example also demonstrates the use of various virtual file systems (VFS) in a single solution. The example uses different VFS to access the functions for working with the file system and functions for working with the network:

- The `VfsNet` program is used for working with the network.

- The `VfsSdCardFs` program is used to work with the file system.

The `kl.Ntpd` program is included in KasperskyOS Community Edition and is an implementation of an NTP client, which gets time parameters from external NTP servers in the background and passes them to the KasperskyOS kernel.

The `kl.rump.Dhcpcd` program is included in KasperskyOS Community Edition and is an implementation of a DHCP client, which gets the parameters of network interfaces from an external DHCP server in the background and passes them to the virtual file system.

The `CMake` system, which is included with KasperskyOS Community Edition, is used to build and run the example.

Example files

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

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

Building and running example

See Building and running examples section.

To ensure that the `messagebus` example will correctly run in Raspberry Pi, you must do the following after building the example and preparing your bootable SD card:

- Create the `/lib` directory on the bootable SD card if this directory doesn't already exist.
- Open the `build/hdd/lib` directory that was generated when building the example and copy the directory contents to the `/lib` directory on the bootable SD card.

I2c_ds1307_rtc example

This example demonstrates use of the `i2c` driver (Inter-Integrated Circuit) in KasperskyOS.

In this example, the `I2cClient` program uses the `i2c` driver interface.

The client library of the `i2c` driver is statically linked to the `I2cClient` program. The `i2c` driver implementation uses a BSP (Board Support Platform) subsystem for configuring clock frequencies (Clocks) and pins multiplexing (PinMux). Therefore, to ensure correct operation of the driver, you need to do the following:

- Link the `I2cClient` program to the `i2c_CLIENT_LIB` client library.
- Link the `I2cClient` program to the `bsp_CLIENT_LIB` client library.
- Create an IPC channel between the `I2cClient` program and the `kl.drivers.I2C` driver.
- Create an IPC channel between the `I2cClient` program and the `kl.drivers.BSP` driver.

The `CMake` system, which is included with KasperskyOS Community Edition, is used to build and run the example.

Example files

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

```
/opt/KasperskyOS-Community-Edition-<version>/examples/i2c_ds1307_rtc
```

Building and running example

> This example is intended to run only on Raspberry Pi. For the example to work correctly, you must connect a DS1307Z real-time clock module to the i2c port.

See Building and running examples section.

i2c_bm8563_rtc example

This example demonstrates use of the `i2c` driver (Inter-Integrated Circuit) in KasperskyOS.

In this example, the `Bm8563` program uses the `i2c` driver interface.

The client library of the `i2c` driver is statically linked to the `Bm8563` program. The `i2c` driver implementation uses a BSP (Board Support Platform) subsystem for configuring clock frequencies (Clocks) and pins multiplexing (PinMux). Therefore, to ensure correct operation of the driver, you need to do the following:

- Link the `Bm8563` program to the `i2c_CLIENT_LIB` client library.
- Link the `Bm8563` program to the `bsp_CLIENT_LIB` client library.
- Create an IPC channel between the `Bm8563` program and the `kl.drivers.I2C` driver.
- Create an IPC channel between the `Bm8563` program and the `kl.drivers.BSP` driver.

The `CMake` system, which is included with KasperskyOS Community Edition, is used to build and run the example.

Example files

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

```
/opt/KasperskyOS-Community-Edition-<version>/examples/i2c_bm8563_rtc
```

Building and running the example

> This example is intended to run only on Radxa ROCK 3A.

See Building and running examples section.

iperf_separate_vfs example

This example demonstrates use of the `iperf` library in KasperskyOS.

In this example, the `Server` program uses the `iperf` library.

By default, the example uses network software emulation (SLIRP) in QEMU. If you configured TAP interfaces for QEMU, you need to change the network settings for starting QEMU (`QEMU_FLAGS` variable) in the `einit/CMakeLists.txt` file to make sure that the example works correctly (for more details, see the comments in the file).

The example does not use DHCP, therefore the IP address of the network interface must be manually indicated in the code of the `Server` program (`server/src/main.cpp`). SLIRP uses the default values.

The `iperf` library in the example is used in server mode. To connect to this server, install the `iperf3` program on the host machine and run it by using the `iperf3 -c localhost` command. If you configured TAP interfaces, indicate the current IP address instead of `localhost`.

The first startup of the example may take a long time because the `iperf` client uses `/dev/urandom` to fill packets with random data. To avoid this, run the `iperf` client with the `--repeating-payload` parameter.

The `CMake` system, which is included with KasperskyOS Community Edition, is used to build and run the example.

Example files

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

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

Building and running example

See Building and running examples section.

Uart example

Example use of the UART driver.

This example shows how to print "Hello World!" to the appropriate port using the UART driver.

A full description of the UART driver interface is provided in the file `/opt/KasperskyOS-Community-Edition-<version>/sysroot-*-kos/include/uart/uart.h`.

Example files

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

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

Building and running example

See Building and running examples section.

spi_check_regs example

This example demonstrates use of the `SPI` (Serial Peripheral Interface) driver in KasperskyOS.

The example shows how to work with the SPI interface on the Sense HAT add-on board for Raspberry Pi. In this example, the `Client` program uses the `SPI` driver interface. The program opens an SPI channel, displays its parameters and sets the necessary operating mode. Then the program sends a data sequence over this channel and waits to receive the ID of the ATTiny controller installed on the Sense HAT board.

The client library of the `SPI` driver is statically linked to the `Client` program. The `Client` program also uses the `gpio` driver to set the controller operating mode and the BSP (Board Support Platform) subsystem for configuring clock frequencies (Clocks) and pins multiplexing (PinMux).

The `CMake` system, which is included with KasperskyOS Community Edition, is used to build and run the example.

Example files

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

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

Building and running example

> For the example to work correctly, you must connect the Sense HAT module to the SPI port.

See Building and running examples section.

barcode_scanner example

This example demonstrates use of a `USB` (Universal Serial Bus) driver in KasperskyOS using the `libevdev` library.

In this example, the `BarcodeScanner` program uses the `libevdev` library for interaction with a barcode scanner connected to the USB port of Raspberry Pi.

The program waits for signals from the barcode scanner and prints the obtained data to `stderr`.

The `CMake` system, which is included with KasperskyOS Community Edition, is used to build and run the example.

Example files

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

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

Building and running example

> For the example to work correctly, you must connect a barcode scanner running in keyboard emulation mode (such as Zebra Symbol LS2208) to the USB port.

See Building and running examples section.

watchdog_system_reset example

This example demonstrates use of the `Watchdog` driver in KasperskyOS.

In this example, the `Client` program uses the `Watchdog` driver interface to interact with the Watchdog timer as follows:

- Receives the current parameters of the `Watchdog` driver and prints them to `stderr`.
- Changes the default value of the timer to a new value and starts the timer.
- Resets the timer several times.
- Waits for the system to restart when the timer is triggered.

The client library of the `Watchdog` driver is statically linked to the `Client` program.

The `CMake` system, which is included with KasperskyOS Community Edition, is used to build and run the example.

Example files

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

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

Building and running example

> This example is intended to run only on Raspberry Pi.

See Building and running examples section.

shared_libs example

This example demonstrates use of static and dynamic libraries in KasperskyOS.

In the example, the `Client` program performs the following actions:

- Calls a function from the `hello_s` static library.
- Calls a function from the `hello_d1` dynamic library that is linked together with the program and loaded into memory when the process is started.
- Calls a function from the `hello_d2` dynamic library that is loaded into memory when calling the `dlopen()` function of the POSIX interface.

To ensure that dynamic libraries can be shared among different processes, the system program named `BlobContainer` is included in the example.

The `CMake` system, which is included with KasperskyOS Community Edition, is used to build and run the example.

Example files

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

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

Building and running example

See Building and running examples section.

To ensure that the `shared_libs` example will correctly run in Raspberry Pi, you must do the following after building the example and preparing your bootable SD card:

- Create the `/lib` directory on the bootable SD card if this directory doesn't already exist.
- Open the `build/hdd/lib` directory that was generated when building the example and copy the directory contents to the `/lib` directory on the bootable SD card.

pal_tests example

This example demonstrates use of the Policy Assertion Language (PAL) when writing tests for a KasperskyOS-based solution security policy. These tests in PAL let you check the solution security policy even before starting development of the software code based on the formal specifications of solution components. For more details about PAL, see Creating and performing tests for a KasperskyOS-based solution security policy.

Example directory in the SDK

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

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

List of programs

The tests check the security events of the following programs:

- `WebServer` – application that provides the user with access to the `Service` program via the web interface.
- `Service` – application that handles user requests and returns the results of request completion.
- `UserManager` – application that implements user account management and user authorization.

Initialization description

The `init.yaml` file is generated automatically in the example build directory and is not important for understanding this example.

Security policy description

The security policy description files for a KasperskyOS-based solution are located at the path `./einit/src`.

The `security.psl` file contains description of the solution security policy. Through the `use declaration`, this top-level file includes the following parts of a solution security policy description provided by PSL files: `core.psl`, `user_mngr.psl`, `service.psl`, `vfs_net.psl`, and `vfs_sdcard.psl`.

The `pal_tests.psl` file contains sets of tests and is also included in the top-level `security.psl` file through the `use` declaration.

Resources

The `./resources/edl` and `./resources/idl` directories contain the EDL and IDL descriptions for the `WebServer`, `Service` and `UserManager` programs.

Operating scenario

The sets of tests described in the `pal_tests.psl` file are executed to check whether each program from the list complies with the defined security policies. The results of tests are displayed in standard output.

Building and running the example

See Creating and performing tests for a KasperskyOS-based solution security policy.

hello_from_rust example

This example demonstrates how to let your KasperskyOS-based solution include a simple program that was developed based on Rust and built using the Cargo build system and package manager.

Example directory in the SDK

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

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

List of programs

The solution includes the `Hello` application, which prints a message to the standard error output. This program was developed using the Rust language.

Initialization description

The solution initialization description file named `init.yaml` is generated during the solution build process based on the following template:

`./einit/src/init.yaml.in`

Security policy description

The `security.psl` file contains the solution security policy description and is generated based on the following template during the solution build process:

`./einit/src/security.psl.in`

During the solution build, the `@INIT_EXTERNAL_ENTITIES@` macro in the `security.psl.in` template is replaced with a list of system programs that are included in the KasperskyOS SDK. For more details, refer to security.psl.in template.

The solution security policy description in this example allows any interaction between different processes and between processes and the kernel. This solution security policy is used to simplify the example. It would be unacceptable to apply such a policy in a real-world solution.

Resources

The directory `./resources` contains an EDL description of the `Hello` program.

The directory `./vendor` contains the libraries and their metadata required for managing dependencies in Rust-based projects that are built using the Cargo build system.

Operating scenario

The `Hello` program prints the message `Hello, world!` to standard error.

Building and running the example

The CMake build system from the KasperskyOS SDK is used during the solution build. The `Hello` program is built using the Cargo build system, which is also included in the SDK. During the solution build process, the `add_custom_command()` command in the `./hello/CMakeLists.txt` file executes the command `cargo build`, which builds the executable file of the `Hello` program. This file is included in a solution as an imported executable file via the `add_executable()` command.

See Building and running examples section.

hello_corrosion example

This example demonstrates how to let your solution include a simple program that was developed based on Rust and uses Corrosion, which is a set of libraries for integrating Rust into CMake projects.

Example directory in the SDK

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

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

List of programs

The solution includes the `Hello` application, which prints a message to the standard error output. This program was developed using the Rust language.

Initialization description

The solution initialization description file named `init.yaml` is generated during the solution build process based on the following template:

`./einit/src/init.yaml.in`

During the example build, the `@INIT_hello_ENTITY_CONNECTIONS@` macro in the `init.yaml` file is replaced with the list of IPC channels containing all system programs that are linked to the `Hello` application.

During the build, the `@INIT_EXTERNAL_ENTITIES@` macro in the `init.yaml.in` template in the `init.yaml` file is replaced by a list of system programs that are linked to applications. This list contains the IPC channels of system programs, the run parameters of the `main()` function, and the values of environment variables.

For more details, see init.yaml.in template.

Security policy description

The `security.psl` file contains the solution security policy description and is generated based on the following template during the solution build process:

`./einit/src/security.psl.in`

During the solution build, the `@INIT_EXTERNAL_ENTITIES@` macro in the `security.psl.in` template is replaced with a list of system programs that are included in the KasperskyOS SDK. For more details, refer to security.psl.in template.

The solution security policy description in this example allows any interaction between different processes and between processes and the kernel. This solution security policy is used to simplify the example. It would be unacceptable to apply such a policy in a real-world solution.

Resources

The directory `./resources` contains an EDL description of the `Hello` program.

Operating scenario

The `Hello` program prints the message `Hello, world!` to standard error.

Building and running the example

The CMake build system from the KasperskyOS SDK is used during the solution build.

To synchronize the target platform for building Rust-based source code with the target platform of the C compiler, you must add the following command to the `./CMakeList.txt` file:

```
set (Rust_CARGO_TARGET ${CMAKE_C_COMPILER_TARGET})
```

To use Corrosion in your solution, you must add the following command to this same file:

```
find_package (Corrosion REQUIRED)
```

To integrate Rust-based source code into your solution, you must call the following function in the `./hello/CMakeList.txt` file:

```
corrosion_import_crate (MANIFEST_PATH ${CMAKE_SOURCE_DIR}/hello/Cargo.toml)
```

This function compiles the `hello` executable file that will be included in the solution.

See Building and running examples section.

mass_storage example

This example demonstrates use of the UsbMassStorage driver in KasperskyOS-based solutions to work with an external USB drive connected to the USB port of the Raspberry Pi 4 B.

Example directory in the SDK

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

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

List of programs

- `Client` is an application that mounts file systems to specified directories on an external USB drive and performs basic file operations (create, write, read, and delete).
- `FileVfs` is an application that implements the server portion of the virtual file system, thereby providing the interface used for interaction with file systems and block devices.
- `UsbMassStorage` is a driver used to access data on external USB drives (USB flash drives and external hard drives).
- `EntropyEntity` – system program that implements random number generation.
- `USB` – driver for managing all types of USB devices.
- `PCIE` is a PCIe bus driver.
- `Bcm2711MboxArmToVc` – driver for working with the VideoCore (VC6) coprocessor via mailbox technology for Raspberry Pi 4 B.

Initialization description

The solution initialization description file named `init.yaml` is generated during the solution build process based on the following template:

`./einit/src/init.yaml.in`

When building the example, the `@INIT_Client_ENTITY_CONNECTIONS+@` macro in the `init.yaml.in` template in the `init.yaml` file is replaced by a list of IPC channels with all the system programs that are linked to the `Client program.` This list serves as a supplement to the manually defined IPC channels in the `init.yaml.in` template.

When building the example, the `@INIT_FileVfs_ENTITY_CONNECTIONS@` macro in the `init.yaml` file is replaced by a list of IPC channels with all the system programs that are linked to the `FileVfs` program.

During the build, the `@INIT_EXTERNAL_ENTITIES@` macro in the `init.yaml.in` template in the `init.yaml` file is replaced by a list of system programs that are linked to applications. This list contains the IPC channels of system programs, the run parameters of the `main()` function, and the values of environment variables.

For more details, refer to init.yaml.in template.

Security policy description

The `security.psl` file contains the solution security policy description and is generated based on the following template during the solution build process:

`./einit/src/security.psl.in`

During the solution build, the `@INIT_EXTERNAL_ENTITIES@` macro in the `security.psl.in` template is replaced with a list of system programs that are included in the KasperskyOS SDK. For more details, refer to security.psl.in template.

Resources

The directory `./resources/edl` contains EDL descriptions for the `Client` and `FileVfs` programs.

Operating scenario

The `Client` program sends requests to the `FileVfs` program via IPC to mount file systems (`ext2`, `ext3`, `ext4`) to the specified directories and perform basic operations with files (create, write, read, and delete) to verify that the file systems are mounted correctly on an external USB drive. The results of operations are displayed in standard output.

Building and running the example

To prepare the USB drive for use in this example, run the following commands:

```
# In the following commands /dev/sde is the block device name of the connected USB drive
parted /dev/sde mklabel msdos
parted /dev/sde mkpart primary ext4 0% 100%
mkfs.ext4 -L flash /dev/sde1
```

See Building and running examples section.

can_loopback example

Example use of the CAN driver.

This example demonstrates use of the CAN driver when there is no need to connect an additional peripheral device for CAN transceivers. In the example, the CAN port is configured and then a CAN packet is sent. The received CAN packet is checked to make sure that it matches the sent packet.

The CAN interface named `can0` is used to configure the port, and the pointer to the port name is assigned via the `CanPortName` variable. The CAN driver also supports the names `can1` and `can2`, which can be used in the test application.

The CAN driver supports multiple operating modes and may work in Classical CAN mode or CANFD mode. CANFD mode is indicated by the `CAN_FLAGS_FD` flag, which is passed to the function for opening the CAN port of the test example. If the `CAN_FLAGS_FD` flag is absent from the settings for opening the port, Classical CAN mode will be used with the corresponding speed limitations.

In the example, the port is opened with a set transfer rate of 800 KBps. The constant `CAN_BR_800KBS` is used for this purpose.

When using the `CAN_FLAGS_FD` flag, the following data transfer rate constants are supported:

- `CANFD_BR_500KBS_D_2MBS`
- `CANFD_BR_500KBS_D_2_5MBS`
- `CANFD_BR_1MBS_D_2MBS`
- `CAN_BR_1MBS`

- `CAN_BR_800KBS`
- `CAN_BR_500KBS`
- `CAN_BR_250KBS`
- `CAN_BR_125KBS`
- `CAN_BR_62_5KBS`

Classical CAN mode cannot use speeds higher than 1 MBps, therefore only the following constants can be used to configure the transfer rate if the `CAN_FLAGS_FD` flag is absent:

- `CAN_BR_1MBS`
- `CAN_BR_800KBS`
- `CAN_BR_500KBS`
- `CAN_BR_250KBS`
- `CAN_BR_125KBS`
- `CAN_BR_62_5KBS`

The CAN test packet contains the ID, length of the data field, and the data. A description of the fields is provided in the `CanFrame` structure. The length of the data field is limited to 8 bytes for Classical CAN, while the length is limited to 64 bytes for CANFD.

Example files

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

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

Building and running the example

> This example is intended to run only on Radxa ROCK 3A.

If you build and run this example in QEMU, an error will occur. This is the expected behavior, because there is no CAN driver for QEMU.

See Building and running examples section.

can2can example

Example use of the CAN driver.

This example demonstrates use of the CAN driver to forward a message between two CAN interfaces.

In contrast to the can_loopback example, which uses only one CAN interface and does not require additional equipment, this example requires two transceivers that are connected to the `CAN_H` and `CAN_L` lines. A description of the ways to connect transceivers to the built-in CAN controllers is provided in the example directory.

The example sends a CAN packet between the `can0` and `can1` interfaces. The example uses the Classical CAN settings, which assumes a transfer rate limit of 1 MBps.

When configuring the transfer rate, you must use only the following constants:

- `CAN_BR_1MBS`
- `CAN_BR_800KBS`
- `CAN_BR_500KBS`
- `CAN_BR_250KBS`
- `CAN_BR_125KBS`
- `CAN_BR_62_5KBS`

The CAN driver does not impose restrictions on the direction of message transmission. In the example, a message is sent from `can0` to `can1`, but you can also send it in the reverse direction when using the `CanTransmitFrame` function.

The test CAN packet contains the packet ID, length of the data field, and the data. A description of the fields is provided in the `CanFrame` structure. For Classical CAN, the length of the data field is limited to 8 bytes.

Example files

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

```
/opt/KasperskyOS-Community-Edition-<version>/examples/can2can
```

Building and running the example

> This example is intended to run only on Radxa ROCK 3A.

If you build and run this example in QEMU, an error will occur. This is the expected behavior, because there is no CAN driver for QEMU.

See Building and running examples section.

adc_hello example

This example demonstrates use of the Sensors driver in KasperskyOS.

In the example, the `AdcHello` program lets you check the functionality of ADC channels. The number of the channel used during the check is defined in the `ADC_CHAN_NUM` macro.

The program calculates the voltage supplied to the channel, normalizes it and prints it to standard output. The program also displays a list of ADC channels available for operation (`SARADC_VIN6` and `SARADC_VIN7` channels are in the output, but are not available for operation).

Example files

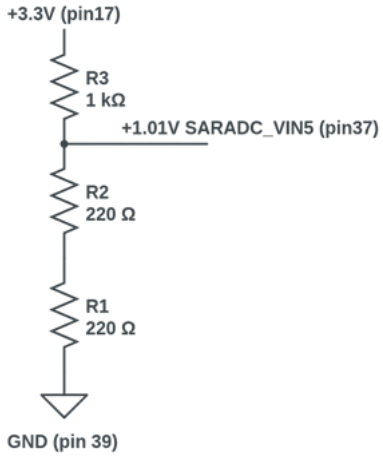
The code of the example and build scripts are available at the following path:

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

Building and running the example

> This example is intended to run only on Radxa ROCK 3A.

For the example to work correctly, you must include a 1 V voltage source to the ADC pin according to the following schematic:



See Building and running examples section.

Information about certain limits set in the system

Header files and IDL files from the KasperskyOS SDK contain constants that set limits in the system (see table below).

Constants that set limits in the system

Subsystem	Constants
`POSIX`	The constants in the `sysroot-*-kos/include/limits.h` file.
`BlobContainer`	The constants in the `sysroot-*-kos/include/kl/EntityLauncher.idl(.h)` files:

	• `MaxArgSize (kl_EntityLauncher_MaxArgSize)` – maximum program launch parameter and environment variable size in bytes. • `MaxArgsCount (kl_EntityLauncher_MaxArgsCount)` – maximum number of launch parameters and environment variables for a program.
`CertificateStorage`	The constants in the `sysroot-*-kos/include/kl/CertificateStorage.idl(.h)` files: • `MaxNumCerts (kl_CertificateStorage_MaxNumCerts)` – maximum number of certificates in a storage. • `MaxCertSize (kl_CertificateStorage_MaxCertSize)` – maximum certificate size in bytes. • `HashSize (kl_CertificateStorage_HashSize)` – size of the certificate storage hash, in bytes.
`Tls`	The constants in the `sysroot-*-kos/include/kl/CertificatePolicy.idl(.h)` files: • `MaxDERCertDataSize (kl_CertificatePolicy_MaxDERCertDataSize)` – maximum DER certificate size in bytes. • `MaxHostAddressBufferSize (kl_CertificatePolicy_MaxHostAddressBufferSize)` – maximum host address buffer size in bytes. The constants in the `sysroot-*-kos/include/kl/crypto/tls/TlsEvent.idl(.h)` files: • `FunctionNameSize (kl_crypto_tls_TlsEvent_FunctionNameSize)` – maximum function name size in bytes. • `IdSize (kl_crypto_tls_TlsEvent_IdSize)` – session identifier size in bytes. • `HostnameSize (kl_crypto_tls_TlsEvent_HostnameSize)` – maximum host name size in bytes. • `PkiEntrySize (kl_crypto_tls_TlsEvent_PkiEntrySize)` – maximum PKI certificate size in bytes. • `MaxCertificatesInChain (kl_crypto_tls_TlsEvent_MaxCertificatesInChain)` – maximum number of certificates in a chain. • `MaxCertificatesInTrustedSet (kl_crypto_tls_TlsEvent_MaxCertificatesInTrustedSet)` – maximum number of trusted certificates. • `KeyFingerprintLength (kl_crypto_tls_TlsEvent_KeyFingerprintLength)` – key fingerprint size in bytes. • `MbedTlsDescriptionSize (kl_crypto_tls_TlsEvent_MbedTlsDescriptionSize)` – maximum MbedTLS error description size in bytes. • `VfsDescriptionSize (kl_crypto_tls_TlsEvent_VfsDescriptionSize)` – maximum VFS error description size in bytes. • `DescriptionSize (kl_crypto_tls_TlsEvent_DescriptionSize)` – maximum event description size in bytes.
`ExecutionManager`	The constants in the `sysroot-*-kos/include/kl/execution_manager/Types.idl(.h)` files: • `NkAppNameMaxSize (kl_execution_manager_Types_NkAppNameMaxSize)` – maximum program name size in bytes. • `NkPathMaxSize (kl_execution_manager_Types_NkPathMaxSize)` – maximum executable file path size in bytes. • `NkEntityNameMaxSize (kl_execution_manager_Types_NkEntityNameMaxSize)` – maximum process name size in bytes. • `NkEiidMaxSize (kl_execution_manager_Types_NkEiidMaxSize)` – maximum process class name size in bytes. • `NkTaskNameMaxSize (kl_execution_manager_Types_NkTaskNameMaxSize)` – maximum process name size in bytes. • `NkArgMaxLen (kl_execution_manager_Types_NkArgMaxLen)` – maximum program launch parameter size in bytes.

	• NkEnvMaxLen (kl_execution_manager_Types_NkEnvMaxLen) – maximum environment variable size in bytes. • NkArgsArrayMaxSize (kl_execution_manager_Types_NkArgsArrayMaxSize) – maximum number of program launch parameters. • NkEnvsArrayMaxSize (kl_execution_manager_Types_NkEnvsArrayMaxSize) – maximum number of environment variables for a program.
KlogStorage	The constants in the sysroot-*-kos/include/kl/KlogStorage.idl(.h) files: • StringSize (kl_KlogStorage_StringSize) – maximum message size in bytes. • MaxMessages (kl_KlogStorage_MaxMessages) – maximum number of messages.
Env	The constants in the sysroot-*-kos/include/kl/Env.idl(.h) files: • MaxArgsCount (kl_Env_MaxArgsCount) – maximum number of launch parameters and environment variables for a program. • MaxArgSize (kl_Env_MaxArgSize) – maximum program launch parameter and environment variable size in bytes. • MaxNameSize (kl_Env_MaxNameSize) – maximum process name size in bytes.
VFS	The constants in the sysroot-*-kos/include/kl/VfsTypes.idl(.h) files: • MaxBytesCount (kl_VfsTypes_MaxBytesCount) – maximum VFS data transmission buffer size in bytes. • MaxPathSize (kl_VfsTypes_MaxPathSize) – maximum path size in bytes. • MaxDevnameSize (kl_VfsTypes_MaxDevnameSize) – maximum device name size in bytes. • MaxFstypeSize (kl_VfsTypes_MaxFstypeSize) – maximum file system name size in bytes. • MaxFsDataSize (kl_VfsTypes_MaxFsDataSize) – maximum data size for the data parameter of the mount() function, in bytes. • MaxFcntlTSize (kl_VfsTypes_MaxFcntlTSize) – maximum data size for an optional parameter of the fcntl() function, in bytes. • MaxIoctlTSize (kl_VfsTypes_MaxIoctlTSize) – maximum data size for an optional parameter of the ioctl() function, in bytes. • MaxSockAddrSize (kl_VfsTypes_MaxSockAddrSize) – maximum IP address size in bytes. • MaxSockOptionSize (kl_VfsTypes_MaxSockOptionSize) – maximum data size for the option_value parameter of the getsockopt() and setsockopt() functions, in bytes. • MaxHostnameSize (kl_VfsTypes_MaxHostnameSize) – maximum host name size in bytes. • MaxServnameSize (kl_VfsTypes_MaxServnameSize) – maximum data size for the servname parameter of the getaddrinfo() function and the service parameter of the getnameinfo() function, in bytes. • MaxMsgNameSize (kl_VfsTypes_MaxMsgNameSize) – maximum data size for the msg_name element of the message parameter of the recvmsg() and sendmsg() functions, in bytes. • MaxMsgDataSize (kl_VfsTypes_MaxMsgDataSize) – maximum data size for a msg_control element of the message parameter of the recvmsg() and sendmsg() functions, in bytes. • MaxIovDataSize (kl_VfsTypes_MaxIovDataSize) – maximum number of the buffer described by the iovec structure in the message parameter of the recvmsg() and sendmsg() functions, and in the iov parameter of the readv() and writev() functions, in bytes. • MaxIovecsCount (kl_VfsTypes_MaxIovecsCount) – maximum number of iovec structures in the message parameter of the recvmsg() and sendmsg() functions, and in the iov parameter of the readv() and writev() functions. • MaxAddrinfoSize (kl_VfsTypes_MaxAddrinfoSize) – maximum data size for the res parameter of the getaddrinfo() function, in bytes.

	• `VfsHostent (kl_VfsTypes_MaxHostentSize)` – maximum data size for a return value of the `gethostbyname()` function, in bytes. • `VfsDnsName (kl_VfsTypes_MaxDnsNameSize)` – maximum data size for the `name` parameter of the `getnetbyname()` function, in bytes. • `MaxProtoentNameSize (kl_VfsTypes_MaxProtoentNameSize)` – maximum protocol name size in the `name` parameter of the `getprotobyname()` function, and in the return value of the `getprotobyname()` and `getprotobynumber()` functions, in bytes. • `MaxProtoentAliasesSize (kl_VfsTypes_MaxProtoentAliasesSize)` – maximum protocol alias size in the return values of the `getprotobyname()` and `getprotobynumber()` functions, in bytes.
`MessageBus`	The constants in the `sysroot-*-kos/include/kl/MessageBusTypes.idl(.h)` files: • `MaxStringLength (kl_MessageBusTypes_MaxStringLength)` – maximum message size in bytes.
`Dhcpcd`	The constants in the `sysroot-*-kos/include/kl/rump/DhcpcdConfig.idl(.h)` files: • `MaxDhcpcdStrSize (kl_rump_DhcpcdConfig_MaxDhcpcdStrSize)` – maximum size of a parameter set received from a DHCP server, in bytes.

Licensing

The *End User License Agreement* is a legally binding agreement between you and AO Kaspersky that stipulates the terms on which you may use KasperskyOS Community Edition.

> Please carefully read the terms of the End User License Agreement before you begin using KasperskyOS Community Edition.

You can view the terms of the End User License Agreement (EULA) in the following ways:

- Read the text of the End User License Agreement before downloading the KasperskyOS Community Edition distribution package.
- Read the document named EULA.<language code>.txt located in the directory `/opt/KasperskyOS-Community-Edition-<version>` after installing KasperskyOS Community Edition.

You accept the terms of the End User License Agreement by selecting the **I agree** check box under the text of the End User License Agreement before downloading the KasperskyOS Community Edition distribution package.

If you do not accept the terms of the End User License Agreement, you must cancel the download of the distribution package and must not use KasperskyOS Community Edition.

Data provision

KasperskyOS Community Edition versions

KasperskyOS Community Edition is distributed in two different versions:

- Version that can be downloaded from the Russian-language website https://os.kaspersky.ru/development.
- Version that can be downloaded from the English-language website https://os.kaspersky.com/development.

Versions differ in the contents of their End User License Agreement, file and differ in the specific information that they automatically transmit to Kaspersky servers when a solution is built using CMake libraries from the SDK.

Data provision in KasperskyOS Community Edition

The version of KasperskyOS Community Edition downloaded from the Russian-language website automatically transmits the following information to Kaspersky servers when a solution build is started:

- Number of the installed version of KasperskyOS Community Edition.
- Unique hardware ID consisting of the checksum of the creation date of the directory /opt/KasperskyOS-Community-Edition-<version>.

The version of KasperskyOS Community Edition downloaded from the English-language website automatically transmits the following information to Kaspersky servers when a solution build is started:

- Number of the installed version of KasperskyOS Community Edition.

In addition to data transmission during the solution build process, both versions also check for the availability of a newer version of KasperskyOS Community Edition.

> If there is no active Internet connection, the solution build occurs without transmitting data or checking for updates.

You can disable the check for SDK updates and transmission of SDK version information to the Kaspersky server by using the `NO_NEW_VERSION_CHECK` parameter of the CMake command `initialize_platform()` during the solution build.

Data is transmitted to account for the number of users of KasperskyOS Community Edition and to obtain information about the distribution and use of KasperskyOS Community Edition.

Any received information is protected by Kaspersky in accordance with the requirements established by law and in accordance with current Kaspersky regulations. Data is transmitted over encrypted communication channels.

Glossary

Address Space Layout Randomization

Address Space Layout Randomization (ASLR) is the use of random addresses for the location of data structures (ELF image, dynamic libraries, stack and heap) in process memory to make it harder to exploit vulnerabilities associated with a conventional process address space structure that is known by a hacker in advance.

Related sections:
Managing processes (low-level API task_api.h)
Managing processes (high-level API task.h)

Affinity mask

An *affinity mask* is a bit mask indicating which specific processors or processor cores must execute a thread.

Related sections:
Managing threads (low-level API thread_api.h)

Application

Program that is designed for interaction with a solution user and for performing user tasks.

Related sections:
Building a KasperskyOS-based solution

Arena chunk descriptor

Structure containing the offset of the arena chunk in bytes (relative to the start of the arena) and the size of the arena chunk in bytes.

Related sections:
Working with an IPC message arena

Arena descriptor

Structure containing three pointers: one pointer to the start of the arena, one pointer to the start of the unused part of the arena, and one pointer to the end of the arena.

Related sections:
Working with an IPC message arena

ASLR

Address Space Layout Randomization

Callable handle

A *callable handle* is a client IPC handle that simultaneously identifies an IPC channel to a server and an endpoint of this server.

Related sections:
Creating handles
Dynamically creating IPC channels using the DCM system program

Capability

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

Related sections:
Resource Access Control
Managing handles (handle_api.h)

CDL

Component Definition Language is a declarative language used to create a formal specification of a solution component.

Related sections:
Formal specifications of KasperskyOS-based solution components
CDL description

Client

In the context of IPC, *client* refers to the client process.

Client library of the solution component

Transport library that converts local calls into IPC requests.

Related sections:
Transport code for IPC

Client process

Process that uses the endpoint of another process via the IPC mechanism. One process can use multiple IPC channels at the same time. A process may act as a client for some IPC channels while acting as a server for other IPC channels.

Related sections:
IPC mechanism

Conditional variable

Synchronization primitive that is used to notify one or more threads about the fulfillment of a condition required by these threads. It is used together with a mutex.

Related sections:
Using synchronization primitives (event.h, mutex.h, rwlock.h, semaphore.h, condvar.h)

Constant part of an IPC message

The part of an IPC message that contains the RIID, MID and (optionally) fixed-size parameters of interface methods.

Related sections:
Overview: IPC message structure
Working with an IPC message arena
IDL data types

Critical section

Section of code in which the resources shared by threads are accessed.

Related sections:
Using synchronization primitives (event.h, mutex.h, rwlock.h, semaphore.h, condvar.h)

DCM handle

A *DCM handle* is an ID used in the API of the system program `DCM` (Dynamic Connection Manager).

Related sections:
Dynamically creating IPC channels using the DCM system program

Describing a security policy for a KasperskyOS-based solution

Set of interrelated text files with the `psl` extension that contain declarations in the PSL language.

Related sections:
Describing a security policy for a KasperskyOS-based solution
General information about a KasperskyOS-based solution security policy description
Security.psl.in template
Example descriptions of basic security policies for KasperskyOS-based solutions
Methods of KasperskyOS core endpoints

Direct memory access

Direct memory access (DMA) is a feature that allows data exchange between devices and the main system memory independently of the processor.

Related sections:
Using DMA (dma.h)
Managing I/O memory isolation (iommu_api.h)

DMA

Direct memory access

DMA buffer

Buffer that consists of one or more physical memory regions (blocks) that are used for direct memory access.

Related sections:
Using DMA (dma.h)
Managing I/O memory isolation (iommu_api.h)
Allocating and freeing memory

EDL

Entity Definition Language is a declarative language used to create a formal specification of a solution component.

Related sections:
Formal specifications of KasperskyOS-based solution components
EDL description

Empty process

A process during the initial stage after its creation when the ELF image of the program is not yet loaded into process memory.

Related sections:
Managing processes (low-level API task_api.h)

Endpoint

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

Related sections:
Overview
IPC mechanism
Methods of KasperskyOS core endpoints

Endpoint interface

Set of signatures for endpoint methods. The endpoint interface is defined in the IDL description.

Related sections:
Formal specifications of KasperskyOS-based solution components
IDL description

Endpoint method

Interface method

Endpoint method ID

An *endpoint method ID* (MID) is the sequence number of the endpoint method within the set of methods of this endpoint (starting at zero).

Related sections:
IPC mechanism
Overview: IPC message structure

Event

Synchronization primitive that is used to notify one or more threads about the fulfillment of a condition required by these threads.

Related sections:
Using synchronization primitives (event.h, mutex.h, rwlock.h, semaphore.h, condvar.h)

Event mask

Value whose bits are interpreted as events that should be tracked or that have already occurred. An event mask has a size of 32 bits and consists of a general part and a specialized part. The common part describes events that are not specific to any resources. The specialized part describes events that are specific to certain resources.

Related sections:
Using notifications (notice_api.h)
Transferring handles

Execute interface

Interface used by the KasperskyOS kernel when querying the Kaspersky Security Module to notify it about kernel startup or about initiating the startup of a process by the kernel or by other processes.

Related sections:
Setting the global parameters of a KasperskyOS-based solution security policy
Binding methods of security models to security events

Formal specification of the KasperskyOS-based solution component

A system of IDL, CDL and EDL descriptions of a solution component (IDL and CDL descriptions are optional).

Related sections:
Formal specifications of KasperskyOS-based solution components

Frozen state of a process

State of a process in which its threads were terminated as a result of an unhandled exception but its resources were not freed, which means that you can get information about this process.

Related sections:
Managing processes (low-level API task_api.h)
Managing processes (high-level API task.h)
Managing threads (high-level API thread.h)
Managing threads (low-level API thread_api.h)

Futex

Low-level synchronization primitive that supports two operations: lock a thread and add it to the futex-linked queue of locked threads, and resume execution of threads from the futex-linked queue of locked threads.

Related sections:
Using futexes (sync.h)
Synchronization endpoint

Global system information

Global system information (GSI) is a structure containing system information, such as the timer count since the KasperskyOS kernel started, the timer counts per second, the number of processors (processor cores) in active state, and processor cache data.

Related sections:
Managing processes (low-level API task_api.h)

GSI

Global system information

Handle

A *handle* is an identifier of a resource (for example, a memory area, port, network interface, or IPC channel). The handle of an IPC channel is called an IPC handle.

Related sections:
Resource Access Control
Managing handles (handle_api.h)

KasperskyOS also uses the following descriptors:

- DCM handle
- Arena descriptor
- Arena chunk descriptor

Handle dereferencing

Operation in which the client sends a handle to the server, and the server receives a pointer to the resource transfer context, the permissions mask of the sent handle, and the ancestor of the handle sent by the client and already owned by the server. Dereferencing occurs when a client that called methods for working with a resource (such as read/write or access closure) sends the server the handle that was received from this server when access to the resource was opened.

Related sections:
Managing handles (handle_api.h)
Dereferencing handles

Handle inheritance tree

Hierarchy of generated resource handles stored in the KasperskyOS kernel.

Related sections:
Managing handles (handle_api.h)

Handle permissions mask

Value whose bits are interpreted as access rights to the resource that is identified by the specific handle.

Related sections:
Resource Access Control
Handle permissions mask
Managing handles (handle_api.h)

Handle transport container

Structure consisting of the following three fields: handle field, handle permissions mask field, and the resource transfer context field. It is used to transfer handles via IPC.

Related sections:
Transferring handles
OCap usage example

Hardware interrupt

Signal sent from a device to direct the processor to immediately pause execution of the current program and instead handle an event related to this device. For example, pressing a key on the keyboard invokes a hardware interrupt that ensures the required response to this pressed key (for example, input of a character).

Related sections:
Managing interrupt processing (irq.h)

IDL

Interface Definition Language is a declarative language used to create a formal specification of a solution component.

Related sections:
Formal specifications of KasperskyOS-based solution components
IDL description

Inferior

Object used in the GDB debugger to work with a process.

Related sections:
Debugging programs using the GDB server of the KasperskyOS kernel

Init description

An init description consists of a text file containing data in YAML format that identifies the processes and IPC channels that are created when the solution starts. The init description file is normally named `init.yaml`.

Related sections:
Overview: Einit and init.yaml
Example init descriptions
Init.yaml.in template

Initializing program

The `Einit` program, which is started by the KasperskyOS kernel, starts other programs according to the init description and creates IPC channels.

Related sections:
Overview: Einit and init.yaml
CMakeLists.txt file for building the Einit program
Structure and startup of a KasperskyOS-based solution
einit

Interface method

Subprogram that is called via IPC.

Related sections:
IPC mechanism
IDL description
Methods of KasperskyOS core endpoints

Interprocess communication

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

Related sections:
IPC
Initializing IPC transport for interprocess communication and managing IPC request processing (transport-kos.h, transport-kos-dispatch.h)
POSIX support limitations

IPC

Interprocess communication

IPC channel

KasperskyOS kernel object that allows processes to interact with each other by transmitting IPC messages. An IPC channel has a client side and a server side, which are identified by a client and server IPC handle, respectively.

Related sections:
IPC mechanism
Creating IPC channels

IPC handle

An *IPC handle* is a handle that identifies an IPC channel. A client IPC handle is necessary for executing a `Call()` system call. A server IPC handle is necessary for executing the `Recv()` and `Reply()` system calls. The callable handle and listener handle are IPC handles.

Related sections:
IPC mechanism
Creating handles
Creating IPC channels
Initializing IPC transport for interprocess communication and managing IPC request processing (transport-kos.h, transport-kos-dispatch.h)
Dynamically creating IPC channels (cm_api.h, ns_api.h)

IPC message

Data packet that is transmitted between different processes and between processes and the KasperskyOS kernel for IPC. An IPC message contains a constant part and an (optional) arena.

Related sections:
Overview: IPC message structure

IPC message arena

Optional part of an IPC message that contains variable-size parameters of interface methods (and/or elements of these parameters).

Related sections:
Overview: IPC message structure
Working with an IPC message arena

IPC request

IPC message sent to a server from a client.

Related sections:
IPC mechanism

IPC response

IPC message sent to a client from a server.

Related sections:
IPC mechanism

IPC transport

Add-on that works on top of system calls for sending and receiving IPC messages and works separately with the constant part and arena of IPC messages. Transport code works on top of this add-on.

Related sections:
Initializing IPC transport for interprocess communication and managing IPC request processing (transport-kos.h, transport-kos-dispatch.h)
Initializing IPC transport for querying the security module (transport-kos-security.h)

KasperskyOS

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

Related sections:
Overview

KasperskyOS security model

Framework for implementing security policies for solutions.

Related sections:
Describing a security policy for a KasperskyOS-based solution
KasperskyOS security models

KasperskyOS-based solution

System software (including the KasperskyOS kernel and Kaspersky Security Module) and applications integrated to work as part of a software/hardware system.

Related sections:
Overview
Structure and startup of a KasperskyOS-based solution
Building a KasperskyOS-based solution

KasperskyOS-based solution component

Program included in a solution.

Related sections:
Overview
Formal specifications of KasperskyOS-based solution components

KPA package

A *KPA package* is a file in the proprietary KPA format that serves as packaging for a program intended for installation in a KasperskyOS-based solution. It includes the following elements:

- KPA package header. Unique sequence of bytes used to identify the KPA format.
- KPA package manifest. Data structure that describes a JSON file containing detailed information about the KPA package.
- KPA package components. Aligned byte sequences with arbitrary contents. KPA package components may include executable files, libraries, text data, or any other data required for the program to work.
- KPA package index. Data structure that describes the number of KPA package components, their checksums, and their sizes.

Related sections:
Working with KPA packages
KPA package manifest

KPA package manifest

A *KPA package manifest* is a JSON file that contains the information required when installing and using a KPA package.

Related sections:
Working with KPA packages
KPA package manifest

KSM

The Kaspersky Security Module is the KasperskyOS kernel module that allows or denies interaction between different processes and between processes and the kernel, and handles queries of processes via the security interface.

Related sections:
IPC control
Resource Access Control

KSS

Kaspersky Security System technology lets you implement solution security policies. This technology prescribes the creation of formal specifications of solution components and descriptions of solution security policies using security models.

Related sections:
Overview
Developing security policies

Listener handle

A *listener handle* is a server IPC handle with extended rights that allow it to add IPC channels to the set of IPC channels identified by this handle.

Related sections:
Creating IPC channels
Creating handles
Initializing IPC transport for interprocess communication and managing IPC request processing (transport-kos.h, transport-kos-dispatch.h)
Dynamically creating IPC channels (cm_api.h, ns_api.h)

MDL

Memory descriptor list

MDL buffer

Buffer that consists of one or more physical memory regions that can be mapped to the memory of multiple processes at the same time. A memory descriptor list is used to map the MDL buffer to process memory.

Related sections:
Creating shared memory
Preparing ELF image segments to be loaded into process memory
Allocating and freeing memory

Memory barrier

A *memory barrier* is an instruction for a compiler or processor that guarantees that memory access operations specified in source code before setting a barrier will be executed before the memory access operations specified in source code after setting a barrier.

Related sections:
Using memory barriers (barriers.h)

Memory Descriptor List

A *memory descriptor list* (MDL) is a data structure containing the addresses and sizes of physical memory regions comprising the MDL buffer.

Related sections:
Creating shared memory

Message signaled interrupt

Message signaled interrupt (MSI) is a hardware interrupt that occurs when the device accesses the interrupt controller via MMIO memory.

Related sections:
Managing interrupt processing (irq.h)

MID

Endpoint method ID

MMIO

Memory-Mapped Input-Output.

Related sections:
Memory-mapped I/O (mmio.h)

MMIO memory

Physical addresses that are mapped to registers and memory of devices. (Portions of one physical address space are used for physical memory addressing and access to registers and memory of devices.)

Related sections:
Memory-mapped I/O (mmio.h)
Allocating and freeing memory

Mutex

A synchronization primitive that provides for mutually exclusive execution of critical sections.

Related sections:
Using synchronization primitives (event.h, mutex.h, rwlock.h, semaphore.h, condvar.h)
POSIX support limitations

Notification receiver

KasperskyOS kernel object that collects notifications about events connected with resources.

Related sections:
Using notifications (notice_api.h)
Managing handles (handle_api.h)

OCap

Object Capability is a security mechanism that is based on capabilities.

Related sections:
Resource Access Control
Managing handles (handle_api.h)

Operating performance point

Operating performance point (OPP) is a combination of the matching frequency and voltage for a processor group.

Related sections:
CPU frequency management endpoint

PAL

Policy Assertion Language is a declarative language used to create solution security policy tests.

Related sections:
Creating and performing tests for a KasperskyOS-based solution security policy
Examples of tests for KasperskyOS-based solution security policies

PCB

Process control block

Process

A running program that has the following distinguishing characteristics:

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

Related sections:
Overview
Starting processes
Init.yaml.in template
Managing processes (high-level API task.h)
Managing processes (low-level API task_api.h)

Process control block

A *process control block* (PCB) is a structure containing process information that is used by the KasperskyOS kernel to manage this specific process.

Related sections:
Managing processes (low-level API task_api.h)

Processor mask

Bit mask indicating the set of processors (processor cores). A flag set in the *i*th bit indicates that a processor with the *i* index is included in the set (numbering starts at zero).

Related sections:
Power management (pm_api.h)

Program

Code that is executed within the context of an individual process.

Related sections:
Building a KasperskyOS-based solution

PSL

Policy Specification Language is a declarative language used to create a solution security policy description.

Related sections:
Describing a security policy for a KasperskyOS-based solution
PSL language syntax

Read-write lock

Synchronization primitive that is used to allow access to resources shared between threads for either write access for one thread or read access for multiple threads at the same time.

Related sections:
Using synchronization primitives (event.h, mutex.h, rwlock.h, semaphore.h, condvar.h)

Recursive mutex

Mutex that can be acquired by a single thread multiple times.

Related sections:
Using synchronization primitives (event.h, mutex.h, rwlock.h, semaphore.h, condvar.h)

Resource

KasperskyOS-based software/hardware system object that can be accessed by processes. Resources can be system resources or user resources.

Related sections:
Resource Access Control

Resource consumer

Process that uses the resources provided by the KasperskyOS kernel or other processes.

Related sections:
Resource Access Control
Managing handles (handle_api.h)
Mic security model

Resource integrity level

Level of trust afforded to a resource. The level of trust in a resource depends on whether this resource was created by a trusted subject within a software/hardware system running KasperskyOS or if it was received from an untrusted external software/hardware system, for example.

Related sections:
Mic security model

Resource provider

Process that manages user resources and manages access to those resources for other processes. For example, drivers are resource providers.

Related sections:
Resource Access Control
Managing handles (handle_api.h)

Resource transfer context

Data that allows the server to identify the resource and its state when access to the resource is requested via descendants of the transferred handle. This normally consists of a data set with various types of data (structure). For example, the transfer context of a file may include the name, path, and cursor position.

Related sections:
Managing handles (handle_api.h)

Resource transfer context object

KasperskyOS kernel object that stores the pointer to the resource transfer context.

Related sections:
Managing handles (handle_api.h)

RIID

Runtime Implementation Identifier

Runtime Implementation Identifier

A *Runtime Implementation Identifier* (RIID) is the sequence number of an endpoint within the set of endpoints of a server (starting at zero).

Related sections:
IPC mechanism
Overview: IPC message structure

SCP

Static connection page

Security audit

A security audit consists of the following sequence of actions. The Kaspersky Security Module provides the KasperskyOS kernel with information about decisions made by this module. Then the kernel forwards this data to the system program `Klog`, which decodes this information and forwards it to the system program `KlogStorage` (data is transmitted via IPC). The latter sends the received audit data to standard output (or standard error) or writes it to a file.

Related sections:
Creating security audit profiles
Examples of security audit profiles
Using the system programs Klog and KlogStorage to perform a security audit

Security audit configuration

Element of a security audit profile that defines the security model objects covered by the security audit and the conditions for performing the security audit.

Related sections:
Creating security audit profiles
Examples of security audit profiles

Security audit data

Information about decisions made by the Kaspersky Security Module, including the actual decisions ("granted" or "denied"), descriptions of security events, results from calling methods of security models, and data on incorrect IPC messages.

Related sections:
Creating security audit profiles

Security audit profile

Set of security audit configurations, each of which defines the security model objects covered by the security audit and the conditions for performing the security audit.

Related sections:
Creating security audit profiles
Binding methods of security models to security events
Examples of security audit profiles
Setting the global parameters of a KasperskyOS-based solution security policy

Security audit runtime-level

The security audit runtime-level is a global parameter of a solution security policy and consists of an unsigned integer that defines the active security audit configuration. (The word "level" here refers to the configuration variant and does not necessarily involve a hierarchy.)

Related sections:
Creating security audit profiles
Setting the global parameters of a KasperskyOS-based solution security policy

Security context

Data that is associated with a security ID and is used by the Kaspersky Security Module to make decisions.

Related sections:
Resource Access Control

Security event

A signal indicating the initiation of communication between a process and another process or between a process and the KasperskyOS kernel.

Related sections:
General information about a KasperskyOS-based solution security policy description
Binding methods of security models to security events
Examples of binding security model methods to security events

Security ID

A *security identifier* (SID) is a globally unique identifier of a resource. The Kaspersky Security Module identifies resources based on their security IDs.

Related sections:
Resource Access Control
Getting a security ID (SID)

Security interface

Interface that is used for interaction between a process and the Kaspersky Security Module. The security interface is defined in the IDL description.

Related sections:
Formal specifications of KasperskyOS-based solution components
EDL description
CDL description
IDL description
Binding methods of security models to security events
Initializing IPC transport for querying the security module (transport-kos-security.h)

Security model expression

Security model method that returns values that can be used as input data for other methods of security models.

Related sections:
General information about a KasperskyOS-based solution security policy description
Binding methods of security models to security events
KasperskyOS security models

Security model method

Element of a security model that determines the permissibility of interactions between various processes and between processes and the KasperskyOS kernel.

Related sections:
General information about a KasperskyOS-based solution security policy description
Binding methods of security models to security events
Examples of binding security model methods to security events
KasperskyOS security models

Security model object

Instance of a class whose definition is a formal description of a security model (in a PSL file).

Related sections:
General information about a KasperskyOS-based solution security policy description
Creating security model objects
KasperskyOS security models

Security model rule

Security model method that returns a "granted" or "denied" decision.

Related sections:
General information about a KasperskyOS-based solution security policy description
Binding methods of security models to security events
Examples of binding security model methods to security events
KasperskyOS security models

Security module decision

A decision on whether to allow or deny a specific interaction between different processes or between a process and the KasperskyOS kernel.

Related sections:
Overview
IPC control
General information about a KasperskyOS-based solution security policy description

Security pattern

A security pattern (or security template) describes a specific recurring security issue that arises in certain known contexts, and provides a well-proven, general scheme for resolving this kind of security issue. A pattern is not a finished project that can be converted directly into code. Instead, it is a solution to a general problem encountered in various projects.

Related sections:
Security patterns for development under KasperskyOS

Security pattern system

Set of security patterns together with instructions on their implementation, combination, and practical use when designing secure software systems.

Related sections:
Security patterns for development under KasperskyOS

Security policy for a KasperskyOS-based solution

Logic for processing security events in the solution. This logic is implemented by the Kaspersky Security Module. The source code of the Kaspersky Security Module is generated from the solution security policy description and formal specifications of solution components.

Related sections:
Overview

Security template

A security template (or security pattern) describes a specific recurring security issue that arises in certain known contexts, and provides a well-proven, general scheme for resolving this kind of security issue. A template is not a finished project that can be converted directly into code. Instead, it is a solution to a general problem encountered in various projects.

Related sections:
Security patterns for development under KasperskyOS

Seed

Starting number of the random number generator (seed) , which determines the sequence of the generated random numbers. In other words, if the same seed value is set, the generator creates identical sequences of random numbers. (The entropy of these numbers is fully determined by the entropy of the seed value, which means that these numbers are not entirely random, but pseudorandom.)

Related sections:
Generating random numbers (random_api.h)
Managing processes (low-level API task_api.h)

Semaphore

Synchronization primitive based on a counter whose value can be atomically changed.

Related sections:
Using synchronization primitives (event.h, mutex.h, rwlock.h, semaphore.h, condvar.h)
POSIX support limitations

Server

In the context of IPC, *server* refers to the server process.

Server library of the solution component

Transport library that converts IPC requests into local calls.

Related sections:
Transport code for IPC

Server process

Process that provides endpoints to other processes via the IPC mechanism. One process can use multiple IPC channels at the same time. A process may act as a server for some IPC channels while acting as a client for other IPC channels.

Related sections:
IPC mechanism

SID

Security ID

SMP

Symmetric Multiprocessing

Related sections:
Debugging programs using the QEMU GDB server
Debugging programs using the GDB server of the KasperskyOS kernel

Static connection page

A *static connection page* (SCP) is a set of structures containing data for creating IPC channels, startup parameters, and environment variables of a program.

Related sections:
Managing processes (low-level API task_api.h)

Subject integrity level

Level of trust afforded to a subject. The trust level of a subject depends on whether the subject interacts with untrusted external software/hardware systems or whether it has a proven standard of quality, for example.

Related sections:
Mic security model

System program

A program that creates the infrastructure for application software (for example, it facilitates hardware operations, supports the IPC mechanism, and implements file systems and network protocols).

Related sections:
Building a KasperskyOS-based solution

System resource

Resource that is managed by the KasperskyOS kernel. Some examples of system resources include processes, memory regions, and interrupts.

Related sections:
Resource Access Control
Managing handles (handle_api.h)

TCB

Thread control block

Thread

A *thread* is an abstraction used to manage the execution of program code. One process can include one or more threads. CPU time is allocated separately for each thread. Each thread may execute the entire code of the program or just a part of the code. The same program code may be executed in multiple threads.

Related sections:
Managing threads (low-level API thread_api.h)
Managing threads (high-level API thread.h)
POSIX support limitations

Thread control block

A *thread control block* (TCB) is a structure containing thread information that is used by the KasperskyOS kernel to manage this specific thread.

Related sections:
Managing threads (low-level API thread_api.h)
Managing threads (high-level API thread.h)

Thread local storage

A *thread local storage* (TLS) is the process memory in which a thread can store data in isolation from other threads.

Related sections:
Managing threads (high-level API thread.h)
Managing threads (low-level API thread_api.h)

TLS

Thread local storage

Transport code

Methods and types in C or C++ for conducting IPC.

Related sections:
Transport code for IPC
Example generation of transport methods and types
Transport code in C++
Generating transport code for development in C++

Transport library

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

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

Related sections:
Transport code for IPC

User resource

Resource that is managed by a process. Examples of user resources: files, input-output devices, data storage.

Related sections:
Resource Access Control
Managing handles (handle_api.h)

User resource context

Data that allows the resource provider to identify the resource and its state when access to the resource is requested by other processes. This normally consists of a data set with various types of data (structure). For example, the context of a file may include the name, path, and cursor position.

Related sections:
Managing handles (handle_api.h)

Information about third-party code

Information about third-party code is contained in the file named legal_notices.txt in the application installation folder.

Trademark notices

Registered trademarks and service marks are the property of their respective owners.

AFS, AIX, AIX 5L, Approach, DYNIX/ptx, IA, IBM, IMS, iSeries, Language Environment, MVS, POWER7, POWER8, PowerPC, RDN, Rhapsody, RS/6000, s3, S/390, Solid, System z, Verilog, VisualAge, z10, z/Architecture, z/OS are trademarks of International Business Machines Corporation, registered in many jurisdictions worldwide.

Aironet, Cisco, IOS are registered trademarks or trademarks of Cisco Systems, Inc. and/or its affiliates in the United States and certain other countries.

AMD, AMD64 are trademarks or registered trademarks of Advanced Micro Devices, Inc.

Android, Chrome, Closure, Dalvik, Dart, Google, Google Chrome, Google Code, Google Test, Motif are trademarks of Google LLC.

Arm and Mbed are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

Aperture, Apple, AppleScript, AppleTalk, Apple TV, Aqua, Carbon, Cocoa, Finder, iMessage, iPhone, Mac, MacBook, Macintosh, macOS, Mac OS, MPW, NetInfo, Newton, Objective-C, OpenCL, OS X, QuickTime, Rosetta, Safari, Sherlock, Tiger, Xcode are trademarks of Apple Inc.

AppLocker, Arc, Convergence, DirectX, Express, Internet Explorer, JScript, Microsoft, Microsoft Edge, MSDN, MS-DOS, MultiPoint, PowerShell, SQL Server, Visual C++, Visual Studio, Win32, Windows, Windows Mobile, Windows Phone, Windows Server, Windows Store, Windows Vista are trademarks of the Microsoft group of companies.

Atom, Core, i960, Intel, Intel386, Intel XScale, Itanium, MCS, MMX, Pentium, SpeedStep, Wireless MMX, Xeon XMM are trademarks of Intel Corporation or its subsidiaries.

BACnet is a registered trademark of the American Society of Heating, Refrigerating and Air-Conditioning Engineers, Inc.

DICOM is the registered trademark of the National Electrical Manufacturers Association for its Standards publications relating to digital communications of medical information.

Docker and the Docker logo are trademarks or registered trademarks of Docker, Inc. in the United States and/or other countries. Docker, Inc. and other parties may also have trademark rights in other terms used herein.

Eclipse Mosquitto is a trademark of Eclipse Foundation, Inc.

F5 is a trademark of F5 Networks, Inc. in the U.S. and in certain other countries.

Linux is the registered trademark of Linus Torvalds in the U.S. and other countries.

OpenSSL is a trademark owned by the OpenSSL Software Foundation.

Python is a trademark or registered trademark of the Python Software Foundation.

Raspberry Pi is a trademark of the Raspberry Pi Foundation.

Fedora, Red Hat are trademarks or registered trademarks of Red Hat, Inc. or its subsidiaries in the United States and other countries.

Firebird is a registered trademark of the Firebird Foundation.

Firefox, Mozilla are trademarks of the Mozilla Foundation in the U.S. and other countries.

FreeBSD is a registered trademark of The FreeBSD Foundation.

K9, Symantec are trademarks or registered trademarks of Symantec Corporation or its affiliates in the U.S. and other countries.

MIPS is a trademark or registered trademark of MIPS Technologies, Inc. in the United States and other countries.

Novell is a registered trademark of Novell Enterprises Inc. in the United States and other countries.

Nvidia is a registered trademark of NVIDIA Corporation.

Oracle is a registered trademark of Oracle and/or its affiliates.

QT is a trademark or registered trademark of The Qt Company Ltd.

S60 is a trademark or registered trademark of Nokia Corporation.

Texas Instruments is a trademark of Texas Instruments.

Trend Micro is a trademark or registered trademark of Trend Micro Incorporated.

Ubuntu is a registered trademark of Canonical Ltd.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

VxWorks is a registered trademark (®) or service mark (SM) of Wind River Systems, Inc. This product is not affiliated with, endorsed, sponsored, supported by the owners of the third-party trademark mentioned herein.