The seL4® Foundation

https://sel4.foundation

Microkit User Manual (v2.2.0)

Contents

1 Introduction

The seL4 Microkit is a small and simple operating system (OS) built on the seL4 microkernel. Microkit is designed for building system with a *static architecture*. A static architecture is one where system resources are assigned up-front at system initialisation time.

1.1 Purpose

The Microkit is intended to:

- provide a small and simple OS for a wide range of IoT, cyberphysical and other embedded use cases;
- provide a reasonable degree of application portability appropriate for the targeted use cases;
- make seL4-based systems easy to develop and deploy within the target areas;
- leverage seL4's strong isolation properties to support a near-minimal *trusted computing base* (TCB);
- retain seL4's trademark performance for systems built with it;
- be, in principle, amenable to formal analysis of system safety and security properties (although such analysis is beyond the initial scope).

1.2 Overview

A Microkit system is built from a set of individual programs that are isolated from each other, and the system, in *protection domains*. Protection domains can interact by calling *protected procedures* or sending *notifications*.

Microkit is distributed as a software development kit (SDK). The SDK includes the tools, libraries and binaries required to build a Microkit system. The Microkit source is also available which allows you to customize or extend Microkit and produce your own SDK.

To build a Microkit system you will write some programs that use `libmicrokit`. Microkit programs are a little different to a typical process on a Linux-like operating system. Rather than a single `main` entry point, a program has four distinct entry points: `init`, `notified` and, potentially, `protected`, `fault`.

On ARM and RISC-V, the individual programs are combined to produce a single bootable *system image*. The format of the image is suitable for loading by the target board's bootloader.

On x86-64, there are two ELF images involved. One for the seL4 kernel, and one for the initialiser that setups up the Microkit system. These are loaded by a Multiboot 2 compliant bootloader.

The Microkit tool, which is provided as part of the SDK, is used to generate the system image(s).

The Microkit tool takes a *system description* as input. The system description is an XML file that specifies all the objects that make up the system.

Note: Microkit does **not** impose any specific build system; you are free to choose the build system that works best for you.

1.3 Document Structure

The Concepts chapter describes the various concepts that make up Microkit. It is recommended that you familiarise yourself with these concepts before trying to build a system.

The SDK chapter describes the software development kit, including its components and system requirements for use.

The Microkit tool chapter describes the host system tool used for generating a system image from the system description and user-programs.

The libmicrokit chapter describes the interfaces to the Microkit library.

The System Description File chapter describes the format of the system description XML file.

The Board Support Packages chapter describes each of the board support packages included in the SDK.

The Rationale chapter documents the rationale for some of the key design choices of Microkit.

The Internals chapter documents some of the internal details for how Microkit works.

2 Concepts

This chapter describes the key concepts provided by Microkit.

As with any set of concepts there are words that take on special meanings. This document attempts to clearly describe all of these terms, however as the concepts are inter-related it is sometimes necessary to use a term prior to its formal introduction.

- system
- protection domain (PD)
- virtual machine (VM)
- memory region
- channel
- protected procedure
- notification
- interrupt
- fault
- ioport

2.1 System

At the most basic level Microkit provides the platform for running a *system* on a specific board. As a *user* of Microkit you use the platform to create a software system that implements your use case. The system is described in a declarative configuration file, and the Microkit tool takes this system description as an input and produces an appropriate system image that can be loaded on the target board.

The key elements that make up a system are *protection domains*, *memory regions* and *channels*.

2.2 Protection Domains

A **protection domain** (PD) is the fundamental runtime abstraction in Microkit. It is analogous, but very different in detail, to a process on a UNIX system.

A PD provides a thread of control that executes within a fixed virtual address space. The isolation provided by the virtual address space is enforced by the underlying hardware MMU.

The virtual address space for a PD has mappings for the PD's *program image* along with any memory regions that the PD can access. The program image is an ELF file containing the code and data which implements the isolated component.

Microkit supports a maximum of 63 protection domains.

2.2.1 Entry points

Although a protection domain is somewhat analogous to a process, it has a considerably different program structure and life-cycle. A process on a typical operating system will have a `main` function which is invoked by the system when the process is created. When the `main` function returns the process is destroyed.

By comparison a protection domain has up to four entry points:

- `init`, `notified` which are required.
- `protected` which is optional.
- `fault` which is required if the PD has children.

When a Microkit system is booted, all PDs in the system execute the `init` entry point.

A PD will not execute any other entry point until `init` has finished.

If a PD is currently executing an entry point, it will not execute any other entry point until the current entry point has finished.

The `notified` entry point will be invoked whenever the protection domain receives a *notification* on a *channel*. The `protected` entry point is invoked when a PD's *protected procedure* is called by another PD. A PD does not have to provide a protected procedure, therefore the `protected` entry point is optional.

The `fault` entry point is invoked when a PD that is a child of another PD causes a fault. A PD does not have to have child PDs, therefore the `fault` entry point is only required for a parent PD.

These entry points are described in more detail in subsequent sections.

Note: The processing of `init` entry points is **not** synchronised across protection domains. Specifically, it is possible for a high priority PD's `notified` or `protected` entry point to be called prior to the completion of a low priority PD's `init` entry point.

The overall computational model for a Microkit system is a set of isolated components reacting to incoming events.

2.2.2 Scheduling

The PD has a number of scheduling attributes that are configured in the system description:

- priority (0 – 254)
- period (microseconds)
- budget (microseconds)
- passive (boolean)

The budget and period bound the fraction of CPU time that a PD can consume. Specifically, the **budget** specifies the amount of time for which the PD is allowed to execute. Once the PD has consumed its budget, it is no longer runnable until the budget is replenished; replenishment happens once every **period** and resets the budget to its initial value. This means that the maximum fraction of CPU time the PD can consume is $\frac{budget}{period}$.

The budget cannot be larger than the period. A budget that equals the period (aka. a "full" budget) behaves like a traditional time slice: After executing for a full period, the PD is preempted and put at the end of the scheduling queue of its priority. In other words, PDs with equal priorities and full budgets are scheduled round-robin with a time slice defined by the period.

The **priority** determines which of the runnable PDs to schedule. A PD is runnable if one of its entry points has been invoked and it has budget remaining in the current period. Runnable PDs of the same priority are scheduled in a round-robin manner.

Passive determines whether the PD is passive. A passive PD will have its scheduling context revoked after initialisation and then bound instead to the PD's notification object. This means the PD will be scheduled on receiving a notification, whereby it will run on the notification's scheduling context. When the PD receives a *protected procedure* by another PD or a *fault* caused by a child PD, the passive PD will run on the scheduling context of the callee.

2.3 Virtual Machines

A *virtual machine* (VM) is a runtime abstraction for running guest operating systems in Microkit. It is similar to a protection domain in that it provides a thread of control that executes within an

isolated virtual address space.

The main difference between a VM and a PD is that VMs have a higher privilege level such that they may function as a guest operations and have their own user-space programs at a separate exception level.

The virtual machine is always a child of a PD. Exceptions caused by the virtual machine are delivered to the parent PD through the `fault` entry point.

Each virtual machine has a number of 'virtual CPU's associated with it. Each virtual CPU (vCPU) that belongs to a virtual machine has its own thread of execution. A vCPU also has an identifier which is used to know which vCPU caused the invocation of the `fault` entry point.

The parent PD is responsible for starting and managing the virtual machine. Microkit provides the abstractions in order to manage the virtual machine through seL4 but there is typically a non-trivial amount of supporting code/infrastructure to properly start and manage a VM.

To keep the (potentially untrusted) virtual machine isolated from the rest of the system, Microkit enforces that a protection domain can only ever manage a single virtual machine.

2.4 Memory Regions

A *memory region* is a range of memory.

A memory region has a *fixed* physical address if it either:

1. has an explicitly defined physical address, or
2. is a subject of a `setvar` element with a `region_paddr` attribute (See System Description File). In this case the tool will automatically choose a suitable physical address.

For memory regions without a fixed physical address, it is not guaranteed for it to be contiguous in physical memory.

Typically, memory regions with a fixed physical address represent memory-mapped device registers or DMA regions.

Memory regions that are within main memory are zero-initialised.

The size of a memory region must be a multiple of a supported page size. The supported page sizes are architecture dependent. For example, on AArch64 architectures, Microkit support 4KiB and 2MiB pages. The page size for a memory region may be specified explicitly in the system description. If page size is not specified, the largest supported page size is used.

Note: The page size also restricts the alignment of the memory region's physical address. A fixed physical address must be a multiple of the specified page size.

A memory region can be *mapped* into one or more protection domains. The mapping has a number of attributes, which include:

- the virtual address at which the region is mapped in the PD
- caching attributes (mostly relevant for device memory)
- permissions (read, write and execute)

Note: When a memory region is mapped into multiple protection domains, the attributes used for different mappings may vary.

2.4.1 Prefilling

A *memory region* may be prefilled with data from a file at build time by specifying the file's name in the System Description File.

In this case, specifying the memory region's size become optional. If a size isn't specified, the memory region will be sized by the length of the prefill file, rounded up to the smallest page size or the user specified page size.

2.5 Channels

A *channel* enables two protection domains to interact using protected procedures or notifications. Each connects exactly two PDs; there are no multi-party channels.

When a channel is created between two PDs, a *channel identifier* is configured for each PD. The *channel identifier* is used by the PD to reference the channel. Each PD can refer to the channel with a different identifier. For example if PDs **A** and **B** are connected by a channel, **A** may refer to the channel using an identifier of **37** while **B** may use **42** to refer to the same channel.

Note: There is no way for a PD to directly refer to another PD in the system. PDs can only refer to other PDs indirectly if there is a channel between them. In this case, the channel identifier is effectively a proxy identifier for the other PD. So, to extend the prior example, **A** can indirectly refer to **B** via the channel identifier **37**. Similarly, **B** can refer to **A** via the channel identifier **42**.

The system supports a maximum of 63 channels and interrupts per protection domain.

2.5.1 Protected procedures

A protection domain may provide a *protected procedure* (PP) which can be invoked from another protection domain. Up to 64 words of data may be passed as arguments when calling a protected procedure. The protected procedure return value may also be up to 64 words.

When a protection domain calls a protected procedure, the procedure executes within the context of the providing protection domain.

A protected call is only possible if the callee has strictly higher priority than the caller. Transitive calls are possible, and as such a PD may call a *protected procedure* in another PD from a `protected` entry point. However the overall call graph between PDs must form a directed, acyclic graph. It follows that a PD can not call itself, even indirectly. For example, `A calls B calls C` is valid (subject to the priority constraint), while `A calls B calls A` is not valid.

When a protection domain is called, the `protected` entry point is invoked. The control returns to the caller when the `protected` entry point returns.

The caller is blocked until the callee returns. Protected procedures must execute in bounded time. It is intended that a future version of Microkit will enforce this condition through static analysis. In the present version the caller must trust the callee to conform.

In general, PPs are provided by services for use by clients that trust the protection domain to provide that service.

To call a PP, a PD calls `microkit_ppcall` passing the channel identifier and a *message* structure. A *message* structure is returned from this function.

When a PD's protected procedure is invoked, the `protected` entry point is invoked with the channel identifier and message structure passed as arguments. The `protected` entry point must return a message structure.

2.5.2 Notifications

A notification is a (binary) semaphore-like synchronisation mechanism. For example, a PD can *notify* another PD to indicate availability of data in a shared memory region if they share a channel.

To notify another PD, a PD calls `microkit_notify`, passing the channel identifier. When a PD receives a notification, the `notified` entry point is invoked with the appropriate channel identifier passed as an argument.

Unlike protected procedures, notifications can be sent in either direction on a channel regardless of priority.

Note: Notifications provide a mechanism for synchronisation between PDs, however this is not a blocking operation. If a PD notifies another PD, that PD will become scheduled to run (if it is not already), but the current PD does **not** block. Of course, if the notified PD has a higher priority than the current PD, then the current PD will be preempted (but not blocked) by the other PD.

Depending on the scheduling, one PD could notify another multiple times without it being scheduled, resulting in a single execution of the `notified` entry point. For example, if PD A notifies PD B three times on the same channel without PD B ever executing, once PD B is scheduled it would only see one notification and hence only enter `notified` once for that channel.

2.6 Interrupts

Hardware interrupts can be used to notify a protection domain. The system description specifies if a protection domain receives notifications for any hardware interrupt sources. Each hardware interrupt is assigned a channel identifier. In this way the protection domain can distinguish the hardware interrupt from other notifications. A specific hardware interrupt can only be associated with at most one protection domain. It should be noted that once a hardware interrupt has been received, it will not be received again until `microkit_irq_ack` is called. The seL4 kernel will mask the hardware interrupt until it has been acknowledged.

Microkit does not provides timers, nor any *sleep* API. After initialisation, activity in the system is initiated by an interrupt causing a `notified` entry point to be invoked. That notified function may in turn notify or call other protection domains that cause other system activity, but eventually all activity indirectly initiated from that interrupt will complete, at which point the system is inactive again until another interrupt occurs.

2.7 Faults

Faults such as an invalid memory access or illegal instruction are delivered to the seL4 kernel which then forwards them to a designated 'fault handler'. By default, all faults caused by protection domains go to the system fault handler which simply prints out details about the fault in a debug configuration.

When a protection domain is a child of another protection domain, the designated fault handler for the child is the parent protection domain. The same applies for a virtual machine.

This means that whenever a fault is caused by a child, it will be delivered to the parent PD instead of the system fault handler via the `fault` entry point. It is then up to the parent to decide how the fault is handled.

The default system fault handler (aka the monitor) has the highest priority and so will execute and handle faults immediately after they occur. For child PDs that have their faults delivered to another PD, the fault being handled depends on when the parent PD is scheduled.

2.8 I/O Ports

I/O ports are x86 mechanisms to access certain physical devices (e.g. PC serial ports or PCI) using the `in` and `out` CPU instructions. The system description specifies if a protection domain

have access to certain port address ranges. These accesses will be executed by seL4 and the result returned to protection domains.

3 SDK

Microkit is distributed as a software development kit (SDK).

The SDK includes support for one or more *boards*. Three *configurations* are supported for each board: *debug*, *release*, and *benchmark*. See the Configurations section for more details.

The SDK contains:

- Microkit user manual (this document)
- Microkit tool

Additionally, for each supported board configuration the following are provided:

- `libmicrokit.a`
- `initialiser.elf`
- `monitor.elf`
- seL4 kernel image
  - On ARM/RISC-V: `sel4.elf`
  - On x86-64: `sel4_64.elf` (64-bit) and `sel4_32.elf` (32-bit)

On ARM and RISC-V, an additional `loader.elf` is provided, which acts as the system's bootloader.

On x86-64, 32-bit/64-bit variants of the kernel are provided as even on 64-bit platforms (e.g QEMU), a 32-bit kernel is required.

There are also examples provided in the `example` directory.

The Microkit SDK does **not** provide, nor require, any specific build system. The user is free to build their system using whatever build system is deemed most appropriate for their specific use case.

The Microkit tool should be invoked by the system build process to transform a system description (and any referenced program images) into an image file which can be loaded by the target board's bootloader.

The ELF files provided as program images should be standard ELF files and have been linked against the provided `libmicrokit.a`.

3.1 Configurations

3.1.1 Debug

The *debug* configuration includes a debug build of the seL4 kernel to allow console debug output using the kernel's UART driver.

3.1.2 Release

The *release* configuration is a release build of the seL4 kernel and is intended for production builds. The loader, monitor, initialiser and kernel do *not* perform any serial output.

3.1.2.1 Proofs The formal verification of seL4 applies to a specific set of seL4 configuration only. This means that making a *release* build of a Microkit system does not imply that the seL4 kernel being used is verified.

Currently Microkit always uses the MCS configuration of seL4 which is still undergoing verification, scheduled to complete for RISC-V in 2026 and AArch64 in 2027. The design proofs for MCS are done but the work to show that the kernel code conforms to the design is still undergoing.

You can find more information about what verified seL4 configurations do exist here

A roadmap for upcoming verification is available here.

3.1.3 Benchmark

The *benchmark* configuration uses a build of the seL4 kernel that exports the hardware's performance monitoring unit (PMU) to PDs. The kernel also tracks information about CPU utilisation. This benchmark configuration exists due a limitation of the seL4 kernel and is intended to be removed once RFC-16 is implemented.

3.1.4 Multi-core (SMP) configurations

The configurations listed above default to using the uni-core configuration of seL4/Microkit. Currently we do not want to default to using the multi-core configuration of the kernel so separate configurations exist for users wanting to use multi-core.

The multi-core configurations are the same as the configurations listed above, except with the *smp-* prefix. For example, *smp-debug* for the multi-core variant of *debug*.

On ARM/RISC-V, the number of cores available defaults to the number of cores the target board has.

On x86-64, the number of cores available depends on how many are detected at run-time. Right now there is a fixed upper limit, see the board-specific documentation for details.

3.2 System Requirements

The Microkit tool requires Linux (x86-64 or AArch64), macOS (x86-64 or AArch64).

On Linux, the Microkit tool is statically linked and should run on any distribution.

On macOS, the Microkit tool should run on macOS 10.12 (Sierra) or higher.

The Microkit tool does not depend on any additional system binaries.

4 Microkit Tool

The Microkit tool is available in `bin/microkit`.

The Microkit tool takes as input a system description. The format of the system description is described in a subsequent chapter.

Usage:

```
microkit [-h] [-o OUTPUT] [-r REPORT]
          [--capdl-json CAPDL_SPEC] [--image-type {binary,elf,uimage}]
          --board [BOARD] --config CONFIG [--search-path [SEARCH_PATH ...]] system
```

The path to the system description file, board to build the system for, and configuration to build for must be provided.

The search paths provided tell the tool where to find any program images specified in the system description file.

In the case of errors, a diagnostic message shall be output to `stderr` and a non-zero code returned.

In the case of success, a loadable image file and a report shall be produced. The type of image is specified by the `--image-type` argument. The output paths for these can be specified by `-o` and `-r` respectively. The default output paths are `loader.img` and `report.txt`.

The report is a plain text file describing important information about the system. The report can be useful when debugging potential system problems. This report does not have a fixed format and may change between versions. It is not intended to be machine readable.

4.1 Image format

4.1.1 ARM

On ARM, the Microkit tool will produce either a raw binary that is intended to be loaded and jumped or an ELF. By default, the tool will produce a raw binary unless the format is specified using the `--image-type` argument.

For the raw binary format, there is a specified entry point of the image that is documented in the board-specific instructions in this manual, under Board Support Packages. If the image is not loaded at this address it should still work by relocating itself upon boot.

This image is expected to be loaded via a previous bootloader, typically U-Boot using the `go` command for the binary image format and the `bootelf` command for ELFs. Note when using `bootelf` you may need to set the `autostart` environment variable with `setenv autostart yes` to have the image executed after `bootelf`.

See the board-specific instructions for more details.

4.1.2 RISC-V

On RISC-V, the Microkit tool can produce either a raw binary that is intended to be loaded and jumped, a uImage with the Linux header or an ELF. Typically the tool will produce a uImage by default, for board-specific details see Board Support Packages.

For the raw binary format, there is a specified entry point of the image that is documented in the board-specific instructions in this manual, under Board Support Packages. If the image is not loaded at this address it should still work by relocating itself upon boot.

For the uImage format. You may load this image anywhere in memory along with your platform's DTB, then use the `bootm` command in U-Boot. It will copy the loader to the correct location in memory and jump to it.

4.1.3 x86-64

On x86-64, the image format is always ELF, it is expected to be loaded as a Multiboot boot module.

See x86-64 generic support for more details.

5 Language Support

There are native APIs for C/C++ and Rust.

libmicrokit exports a C API and so can be used in any language that supports C FFI.

For Rust, native bindings exist but are not included in the SDK itself. They are available at rust-sel4.

6 libmicrokit

All program images should link against `libmicrokit.a`.

The library provides the C runtime for the protection domain, along with interfaces for the Microkit APIs.

The component must provide the following functions:

```
void init(void);
void notified(microkit_channel ch);
```

If the protection domain provides a protected procedure it must also implement:

```
microkit_msginfo protected(microkit_channel ch, microkit_msginfo msginfo);
```

If the protection domain has children it must also implement:

```
seL4_Bool fault(microkit_child child, microkit_msginfo msginfo,
                microkit_msginfo *reply_msginfo);
```

`libmicrokit` provides the following functions:

```
microkit_msginfo microkit_ppcall(microkit_channel ch, microkit_msginfo msginfo);
void microkit_notify(microkit_channel ch);
microkit_msginfo microkit_msginfo_new(seL4_Word label, seL4_Uint16 count);
seL4_Word microkit_msginfo_get_label(microkit_msginfo msginfo);
seL4_Word microkit_msginfo_get_count(microkit_msginfo msginfo);
void microkit_irq_ack(microkit_channel ch);
void microkit_deferred_notify(microkit_channel ch);
void microkit_deferred_irq_ack(microkit_channel ch);
void microkit_pd_restart(microkit_child pd, seL4_Word entry_point);
void microkit_pd_stop(microkit_child pd);
void microkit_mr_set(seL4_Uint8 mr, seL4_Word value);
seL4_Word microkit_mr_get(seL4_Uint8 mr);
void microkit_vcpu_restart(microkit_child vcpu, seL4_Word entry_point);
void microkit_vcpu_stop(microkit_child vcpu);
void microkit_vcpu_arm_inject_irq(microkit_child vcpu, seL4_Uint16 irq,
                                  seL4_Uint8 priority, seL4_Uint8 group,
                                  seL4_Uint8 index);
void microkit_vcpu_arm_ack_vppi(microkit_child vcpu, seL4_Word irq);
seL4_Word microkit_vcpu_arm_read_reg(microkit_child vcpu, seL4_Word reg);
void microkit_vcpu_arm_write_reg(microkit_child vcpu, seL4_Word reg, seL4_Word value);
void microkit_arm_smc_call(seL4_ARM_SMCContext *args, seL4_ARM_SMCContext *response);
void microkit_x86_ioport_write_8(microkit_ioport ioport_id,
                                 seL4_Word port_addr, seL4_Word data);
void microkit_x86_ioport_write_16(microkit_ioport ioport_id,
                                  seL4_Word port_addr, seL4_Word data);
void microkit_x86_ioport_write_32(microkit_ioport ioport_id,
                                  seL4_Word port_addr, seL4_Word data);
seL4_Uint8 microkit_x86_ioport_read_8(microkit_ioport ioport_id, seL4_Word port_addr);
seL4_Uint16 microkit_x86_ioport_read_16(microkit_ioport ioport_id, seL4_Word port_addr);
seL4_Uint32 microkit_x86_ioport_read_32(microkit_ioport ioport_id, seL4_Word port_addr);
seL4_Word microkit_vcpu_x86_read_vmcs(microkit_child vcpu, seL4_Word field);
void microkit_vcpu_x86_write_vmcs(microkit_child vcpu, seL4_Word field, seL4_Word value);
seL4_Word microkit_vcpu_x86_read_msr(microkit_child vcpu, seL4_Word field);
```

```
void microkit_vcpu_x86_write_msr(microkit_child vcpu, seL4_Word field, seL4_Word value);
void microkit_vcpu_x86_enable_ioport(microkit_child vcpu, microkit_ioport ioport_id,
                                     seL4_Word port_addr, seL4_Word length);
void microkit_vcpu_x86_disable_ioport(microkit_child vcpu,
                                      seL4_Word port_addr, seL4_Word length);
void microkit_vcpu_x86_write_regs(microkit_child vcpu, seL4_VCPUContext *regs);
```

6.1 `void init(void)`

Every PD must expose an `init` entry point. This is called by the system at boot time.

6.2 `void notified(microkit_channel ch)`

The `notified` entry point is called by the system when a PD has received a notification on a channel.

`ch` identifies the channel which has been notified (and indirectly the PD that performed the notification).

Note: `ch` could identify an interrupt.

Channel identifiers are specified in the system configuration.

6.3 `microkit_msginfo protected(microkit_channel ch, microkit_msginfo msginfo)`

The `protected` entry point is optional. This is invoked when another PD calls `microkit_ppcall` on a channel shared with the PD.

The `ch` argument identifies the channel on which the PP was invoked. Indirectly this identifies the PD performing the call. Channel identifiers are specified in the system configuration. **Note:** The channel argument is passed by the system and is unforgeable.

The `msginfo` argument is the argument passed to the PP and is provided by the calling PD. The contents of the message is up to a pre-arranged protocol between the PDs. The message contents are opaque to the system. Note: The message is *copied* from the caller.

The returned `microkit_msginfo` is the return value of the protected procedure. As with arguments, this is *copied* to the caller.

6.4 `seL4_Bool fault(microkit_child child, microkit_msginfo msginfo, microkit_msginfo *reply_msginfo)`

The `fault` entry point being invoked depends on whether the given PD has children. It is invoked when a child PD or VM causes a fault.

The `child` argument identifies the child that caused the fault.

The `msginfo` argument is given by the seL4 kernel when a fault occurs and contains information as to what fault occurred.

The `reply_msginfo` argument is given by libmicrokit and can be used to reply to the fault.

The returned `seL4_Bool` is whether or not to reply to the fault with the message `reply_msginfo`. Returning `seL4_True` will reply to the fault. Returning `seL4_False` will not reply to the fault.

You can use `microkit_msginfo_get_label` on `msginfo` to deduce what kind of fault happened (for example, whether it was a user exception or a virtual memory fault).

Whether or not you reply to the fault depends on the type of fault that has occurred and how you want to handle it.

To find the full list of possible faults that could occur and details regarding to replying to a particular kind of fault, please see the 'Faults' section of the seL4 reference manual.

6.5 `microkit_msginfo microkit_ppcall(microkit_channel ch, microkit_msginfo msginfo)`

Performs a call to a protected procedure in a different PD. The `ch` argument identifies the protected procedure to be called. `msginfo` is passed as argument to the protected procedure. Channel identifiers are specified in the system configuration.

The protected procedure's return data is returned in the `microkit_msginfo`.

6.6 `void microkit_notify(microkit_channel ch)`

Notify the channel `ch`. Channel identifiers are specified in the system configuration.

6.7 `void microkit_irq_ack(microkit_channel ch)`

Acknowledge the interrupt identified by the specified channel.

6.8 `void microkit_deferred_notify(microkit_channel ch)`

The same as `microkit_notify` but will instead not actually perform the notify until the entry point where `microkit_deferred_notify` was called returns.

It is important to note that only a single 'deferred' API call can be made within the same entry point.

The purpose of this API is for performance critical code as this API saves a kernel system call.

6.9 `void microkit_deferred_irq_ack(microkit_channel ch)`

The same as `microkit_irq_ack` but will instead not actually perform the IRQ acknowledge until the entry point where `microkit_deferred_irq_ack` was called returns.

It is important to note that only a single 'deferred' API call can be made within the same entry point.

The purpose of this API is for performance critical code as this API saves a kernel system call.

6.10 `void microkit_pd_restart(microkit_child pd, uintptr_t entry_point)`

Restart the execution of a child protection domain with ID `pd` at the given `entry_point`. This will set the program counter of the child protection domain to `entry_point`.

6.11 `void microkit_pd_stop(microkit_child pd)`

Stop the execution of the child protection domain with ID `pd`.

6.12 `microkit_msginfo microkit_msginfo_new(uint64_t label, uint16_t count)`

Creates a new message structure.

The message can be passed to `microkit_ppcall` or returned from `protected`.

6.13 `uint64_t microkit_msginfo_get_label(microkit_msginfo msginfo)`

Returns the label from a message.

6.14 `uint64_t microkit_msginfo_get_count(microkit_msginfo msginfo)`

Returns the count of message registers in the message.

6.15 `uint64_t microkit_mr_get(uint8_t mr)`

Get a message register.

6.16 `void microkit_mr_set(uint8_t mr, uint64_t value)`

Set a message register.

6.17 `void microkit_vcpu_restart(microkit_child vcpu, seL4_Word entry_point)`

Restart the execution of a VM's virtual CPU with ID `vcpu` at the given `entry point`. This will set the program counter of the vCPU to `entry_point`.

6.18 `void microkit_vcpu_stop(microkit_child vcpu)`

Stop the execution of the VM's virtual CPU with ID `vcpu`.

6.19 `void microkit_vcpu_arm_inject_irq(microkit_child vcpu, seL4_Uint16 irq, seL4_Uint8 priority, seL4_Uint8 group, seL4_Uint8 index)`

Inject a virtual ARM interrupt for a virtual CPU `vcpu` with IRQ number `irq`. The `priority` (0-31) determines what ARM GIC (generic interrupt controller) priority level the virtual IRQ will be injected as. The `group` determines whether the virtual IRQ will be injected into secure world (1) or non-secure world (0). The `index` is the index of the virtual GIC list register.

6.20 `void microkit_vcpu_arm_ack_vppi(microkit_child vcpu, seL4_Word irq)`

Acknowledge a ARM virtual Private Peripheral Interrupt (PPI) with IRQ number `irq` for a VM's vCPU with ID `vcpu`.

6.21 `seL4_Word microkit_vcpu_arm_read_reg(microkit_child vcpu, seL4_Word reg)`

Read a register for a given virtual CPU with ID `vcpu`. The `reg` argument is the index of the register that is read. The list of registers is defined by the enum `seL4_VCPUReg` in the seL4 source code.

6.22 `void microkit_vcpu_arm_write_reg(microkit_child vcpu, seL4_Word reg, seL4_Word value)`

Write to a register for a given virtual CPU with ID `vcpu`. The `reg` argument is the index of the register that is written to. The `value` argument is what the register will be set to. The list of registers is defined by the enum `seL4_VCPUReg` in the seL4 source code.

6.23 `void microkit_arm_smc_call(seL4_ARM_SMCContext *args, seL4_ARM_SMCContext *response)`

The API takes in arguments for a Secure Monitor Call which will be performed by seL4. Any response values will be placed into the `response` structure.

The `seL4_ARM_SMCContext` structure contains fields for registers x0 to x7.

Note that this API is only available when the PD making the call has been configured to have SMC enabled in the SDF. Note that when the kernel makes the actual SMC, it cannot pre-empt the Secure Monitor and therefore any kernel WCET properties are no longer guaranteed.

6.24 `void microkit_x86_ioport_write_(8|16|32)(microkit_ioport ioport_id, seL4_Word port_addr, seL4_Word data)`

Write an 8, 16, or 32 bits value at port address `port_addr` to I/O Port with ID `ioport_id`.

6.25 `seL4_Uint(8|16|32) microkit_x86_ioport_read_(8|16|32)(microkit_ioport ioport_id, seL4_Word port_addr)`

Read an 8, 16, or 32 bits value at port address `port_addr` from I/O Port with ID `ioport_id`.

6.26 `seL4_Word microkit_vcpu_x86_read_vmcs(microkit_child vcpu, seL4_Word field)`

Read a Virtual Machine Control Structure field specified by `field` for a given virtual CPU with ID `vcpu` using the `vmread` instruction.

All the VMCS fields are listed in the seL4 source at `include/arch/x86/arch/object/vcpu.h`.

The following fields can be read from:

- `VMX_GUEST_RIP`
- `VMX_GUEST_RSP`
- `VMX_GUEST_ES_SELECTOR`
- `VMX_GUEST_CS_SELECTOR`
- `VMX_GUEST_SS_SELECTOR`
- `VMX_GUEST_DS_SELECTOR`
- `VMX_GUEST_FS_SELECTOR`
- `VMX_GUEST_GS_SELECTOR`
- `VMX_GUEST_LDTR_SELECTOR`
- `VMX_GUEST_TR_SELECTOR`
- `VMX_GUEST_DEBUGCTRL`
- `VMX_GUEST_PAT`
- `VMX_GUEST_EFER`
- `VMX_GUEST_PERF_GLOBAL_CTRL`
- `VMX_GUEST_PDPTE0`
- `VMX_GUEST_PDPTE1`
- `VMX_GUEST_PDPTE2`
- `VMX_GUEST_PDPTE3`
- `VMX_GUEST_ES_LIMIT`
- `VMX_GUEST_CS_LIMIT`
- `VMX_GUEST_SS_LIMIT`
- `VMX_GUEST_DS_LIMIT`
- `VMX_GUEST_FS_LIMIT`

- VMX_GUEST_GS_LIMIT
- VMX_GUEST_LDTR_LIMIT
- VMX_GUEST_TR_LIMIT
- VMX_GUEST_GDTR_LIMIT
- VMX_GUEST_IDTR_LIMIT
- VMX_GUEST_ES_ACCESS_RIGHTS
- VMX_GUEST_CS_ACCESS_RIGHTS
- VMX_GUEST_SS_ACCESS_RIGHTS
- VMX_GUEST_DS_ACCESS_RIGHTS
- VMX_GUEST_FS_ACCESS_RIGHTS
- VMX_GUEST_GS_ACCESS_RIGHTS
- VMX_GUEST_LDTR_ACCESS_RIGHTS
- VMX_GUEST_TR_ACCESS_RIGHTS
- VMX_GUEST_INTERRUPTABILITY
- VMX_GUEST_ACTIVITY
- VMX_GUEST_SMBASE
- VMX_GUEST_SYSENTER_CS
- VMX_GUEST_PREEMPTION_TIMER_VALUE
- VMX_GUEST_ES_BASE
- VMX_GUEST_CS_BASE
- VMX_GUEST_SS_BASE
- VMX_GUEST_DS_BASE
- VMX_GUEST_FS_BASE
- VMX_GUEST_GS_BASE
- VMX_GUEST_LDTR_BASE
- VMX_GUEST_TR_BASE
- VMX_GUEST_GDTR_BASE
- VMX_GUEST_IDTR_BASE
- VMX_GUEST_DR7
- VMX_GUEST_RFLAGS
- VMX_GUEST_PENDING_DEBUG_EXCEPTIONS
- VMX_GUEST_SYSENTER_ESP
- VMX_GUEST_SYSENTER_EIP
- VMX_CONTROL_CR0_MASK
- VMX_CONTROL_CR4_MASK
- VMX_CONTROL_CR0_READ_SHADOW
- VMX_CONTROL_CR4_READ_SHADOW
- VMX_DATA_INSTRUCTION_ERROR
- VMX_DATA_EXIT_INTERRUPT_INFO
- VMX_DATA_EXIT_INTERRUPT_ERROR
- VMX_DATA_IDT_VECTOR_INFO
- VMX_DATA_IDT_VECTOR_ERROR
- VMX_DATA_EXIT_INSTRUCTION_LENGTH
- VMX_DATA_EXIT_INSTRUCTION_INFO
- VMX_DATA_GUEST_PHYSICAL
- VMX_DATA_IO_RCX
- VMX_DATA_IO_RSI
- VMX_DATA_IO_RDI
- VMX_DATA_IO_RIP
- VMX_DATA_GUEST_LINEAR_ADDRESS
- VMX_CONTROL_ENTRY_INTERRUPTION_INFO

- VMX_CONTROL_PIN_EXECUTION_CONTROLS
- VMX_CONTROL_PRIMARY_PROCESSOR_CONTROLS
- VMX_CONTROL_SECONDARY_PROCESSOR_CONTROLS
- VMX_CONTROL_EXCEPTION_BITMAP
- VMX_CONTROL_ENTRY_CONTROLS
- VMX_CONTROL_EXIT_CONTROLS
- VMX_GUEST_CR0
- VMX_GUEST_CR3
- VMX_GUEST_CR4

6.27 `void microkit_vcpu_x86_write_vmcs(microkit_child vcpu, seL4_Word field, seL4_Word value)`

Write a Virtual Machine Control Structure field specified by `field` for a given virtual CPU with ID `vcpu` using the `vmwrite` instruction. The value may be modified to ensure any bits that are fixed in the hardware are correct, and that any features required for kernel correctness are not disabled.

All the VMCS fields are listed in the seL4 source at `include/arch/x86/arch/object/vcpu.h`.

The following fields can be written to:

- VMX_GUEST_RIP
- VMX_GUEST_RSP
- VMX_GUEST_ES_SELECTOR
- VMX_GUEST_CS_SELECTOR
- VMX_GUEST_SS_SELECTOR
- VMX_GUEST_DS_SELECTOR
- VMX_GUEST_FS_SELECTOR
- VMX_GUEST_GS_SELECTOR
- VMX_GUEST_LDTR_SELECTOR
- VMX_GUEST_TR_SELECTOR
- VMX_GUEST_DEBUGCTRL
- VMX_GUEST_PAT
- VMX_GUEST_EFER
- VMX_GUEST_PERF_GLOBAL_CTRL
- VMX_GUEST_PDPTE0
- VMX_GUEST_PDPTE1
- VMX_GUEST_PDPTE2
- VMX_GUEST_PDPTE3
- VMX_GUEST_ES_LIMIT
- VMX_GUEST_CS_LIMIT
- VMX_GUEST_SS_LIMIT
- VMX_GUEST_DS_LIMIT
- VMX_GUEST_FS_LIMIT
- VMX_GUEST_GS_LIMIT
- VMX_GUEST_LDTR_LIMIT
- VMX_GUEST_TR_LIMIT
- VMX_GUEST_GDTR_LIMIT
- VMX_GUEST_IDTR_LIMIT
- VMX_GUEST_ES_ACCESS_RIGHTS
- VMX_GUEST_CS_ACCESS_RIGHTS
- VMX_GUEST_SS_ACCESS_RIGHTS

- VMX_GUEST_DS_ACCESS_RIGHTS
- VMX_GUEST_FS_ACCESS_RIGHTS
- VMX_GUEST_GS_ACCESS_RIGHTS
- VMX_GUEST_LDTR_ACCESS_RIGHTS
- VMX_GUEST_TR_ACCESS_RIGHTS
- VMX_GUEST_INTERRUPTABILITY
- VMX_GUEST_ACTIVITY
- VMX_GUEST_SMBASE
- VMX_GUEST_SYSENTER_CS
- VMX_GUEST_PREEMPTION_TIMER_VALUE
- VMX_GUEST_ES_BASE
- VMX_GUEST_CS_BASE
- VMX_GUEST_SS_BASE
- VMX_GUEST_DS_BASE
- VMX_GUEST_FS_BASE
- VMX_GUEST_GS_BASE
- VMX_GUEST_LDTR_BASE
- VMX_GUEST_TR_BASE
- VMX_GUEST_GDTR_BASE
- VMX_GUEST_IDTR_BASE
- VMX_GUEST_DR7
- VMX_GUEST_RFLAGS
- VMX_GUEST_PENDING_DEBUG_EXCEPTIONS
- VMX_GUEST_SYSENTER_ESP
- VMX_GUEST_SYSENTER_EIP
- VMX_CONTROL_CR0_MASK
- VMX_CONTROL_CR4_MASK
- VMX_CONTROL_CR0_READ_SHADOW
- VMX_CONTROL_CR4_READ_SHADOW
- VMX_CONTROL_EXCEPTION_BITMAP
- VMX_CONTROL_ENTRY_INTERRUPTION_INFO
- VMX_CONTROL_ENTRY_EXCEPTION_ERROR_CODE
- VMX_CONTROL_PIN_EXECUTION_CONTROLS
- VMX_CONTROL_PRIMARY_PROCESSOR_CONTROLS
- VMX_CONTROL_SECONDARY_PROCESSOR_CONTROLS
- VMX_CONTROL_ENTRY_CONTROLS
- VMX_CONTROL_EXIT_CONTROLS
- VMX_GUEST_CR3
- VMX_GUEST_CR0
- VMX_GUEST_CR4

6.28 `seL4_Word microkit_vcpu_x86_read_msr(microkit_child vcpu, seL4_Word field)`

Read a 64-bit Model Specific Register of a given virtual CPU with ID `vcpu` specified by `field` from hardware using the `rdmsr` instruction. The following registers can be written to:

- IA32_LSTAR_MSR
- IA32_STAR_MSR
- IA32_CSTAR_MSR
- IA32_FMASK_MSR

6.29 `void microkit_vcpu_x86_write_msr(microkit_child vcpu, seL4_Word field, seL4_Word value)`

Write a 64-bit Model Specific Register of a given virtual CPU with ID `vcpu` specified by `field` to hardware using the `wrmsr` instruction. The following registers can be read from:

- `IA32_LSTAR_MSR`
- `IA32_STAR_MSR`
- `IA32_CSTAR_MSR`
- `IA32_FMASK_MSR`

6.30 `void microkit_vcpu_x86_enable_ioport(microkit_child vcpu, microkit_ioport ioport_id, seL4_Word port_addr, seL4_Word length)`

Enable I/O port range `[port_addr..port_addr + length)` of I/O Port with ID `ioport_id` in the execution environment of a given virtual CPU with ID `vcpu`.

6.31 `void microkit_vcpu_x86_disable_ioport(microkit_child vcpu, seL4_Word port_addr, seL4_Word length)`

Disable I/O port range `[port_addr..port_addr + length)` in the execution environment of a given virtual CPU with ID `vcpu`.

6.32 `void microkit_vcpu_x86_write_regs(microkit_child vcpu, seL4_VCPUContext *regs)`

Write the registers of a given virtual CPU with ID `vcpu`. The `regs` argument is the pointer to the struct of registers `seL4_VCPUContext` that are written from.

7 System Description File

This section describes the format of the System Description File (SDF).

The system description file is an XML file that is provided as input to the `microkit` tool.

The root element of the XML file is `system`.

Within the `system` root element the following child elements are supported:

- `protection_domain`
- `memory_region`
- `channel`

7.1 `protection_domain`

The `protection_domain` element describes a protection domain.

It supports the following attributes:

- `name`: A unique name for the protection domain
- `priority`: The priority of the protection domain (integer 0 to 254). Defaults to 0.
- `budget`: (optional) The PD's budget in microseconds; defaults to 1,000.
- `period`: (optional) The PD's period in microseconds; must not be smaller than the budget; defaults to the budget.
- `passive`: (optional) Indicates that the protection domain will be passive and thus have its scheduling context removed after initialisation; defaults to false.
- `stack_size`: (optional) Number of bytes that will be used for the PD's stack. Must be be between 4KiB and 16MiB and be 4K page-aligned. Defaults to 8KiB.
- `cpu`: (optional) set the physical CPU core this PD will run on. Defaults to zero.
- `smc`: (optional, only on ARM) Allow the PD to give an SMC call for the kernel to perform.. Defaults to false.

Additionally, it supports the following child elements:

- `program_image`: (exactly one) Describes the program image for the protection domain.
- `map`: (zero or more) Describes mapping of memory regions into the protection domain.
- `irq`: (zero or more) Describes hardware interrupt associations.
- `setvar`: (zero or more) Describes variable rewriting.
- `protection_domain`: (zero or more) Describes a child protection domain.
- `virtual_machine`: (zero or one) Describes a child virtual machine.
- `ioport`: (zero or more) Describes an I/O port, x86-64 only.

The `program_image` element has the following attributes:

- `path`: path to an ELF file.
- `path_for_symbols`: (optional) path to an ELF that will be used just for searching up and patching symbols rather than the ELF specified in `path`.

The `map` element has the following attributes:

- `mr`: Identifies the memory region to map.
- `vaddr`: Identifies the virtual address at which to map the memory region.
- `perms`: Identifies the permissions with which to map the memory region. Can be a combination of `r` (read), `w` (write), and `x` (eXecute), with the exception of a write-only mapping (just `w`). Defaults to read-write.
- `cached`: (optional) Determines if mapped with caching enabled or disabled. Defaults to `true`.

- `setvar_vaddr`: (optional) Specifies a symbol in the program image. This symbol will be rewritten with the virtual address of the memory region.
- `setvar_size`: (optional) Specifies a symbol in the program image. This symbol will be rewritten with the size of the memory region.
- `setvar_prefill_size`: (optional) Specifies a symbol in the program image. This symbol will be rewritten with the size of the prefilled data.

The `irq` element has the following attributes on ARM and RISC-V:

- `irq`: The hardware interrupt number.
- `id`: The channel identifier. Must be at least 0 and less than 63.
- `trigger`: (optional) Whether the IRQ is edge triggered ("edge") or level triggered ("level"). Defaults to "level".
- `setvar_id`: (optional) Specifies a symbol in the program image. This symbol will be rewritten with the channel identifier of the IRQ.

The `irq` element has the following attributes when registering x86-64 IOAPIC interrupts:

- `id`: The channel identifier. Must be at least 0 and less than 63.
- `pin`: IOAPIC pin that generates the interrupt.
- `vector`: CPU vector to deliver the interrupt to.
- `ioapic`: (optional) Zero based index of the IOAPIC to get the interrupt from. Defaults to 0.
- `trigger`: (optional) Whether the IRQ is edge triggered ("edge") or level triggered ("level"). Defaults to "level".
- `polarity`: (optional) Whether the line polarity is high ("high") or low ("low"). Defaults to "high".
- `setvar_id`: (optional) Specifies a symbol in the program image. This symbol will be rewritten with the channel identifier of the IRQ.

The `irq` element has the following attributes when registering x86-64 MSI interrupts:

- `id`: The channel identifier. Must be at least 0 and less than 63.
- `pcidev`: The PCI device address of the device that will generate the interrupt, in BUS:DEV:FUNC notation (e.g. 01:1f:2).
- `handle`: Value of the handle programmed into the data portion of the MSI.
- `vector`: CPU vector to deliver the interrupt to.
- `setvar_id`: (optional) Specifies a symbol in the program image. This symbol will be rewritten with the channel identifier of the IRQ.

The `ioport` element has the following attributes:

- `id`: The I/O port identifier. Must be at least 0 and less than 63.
- `addr`: The base address of the I/O port.
- `size`: The size in bytes of the I/O port region.
- `setvar_id`: (optional) Specifies a symbol in the program image. This symbol will be rewritten with the I/O port identifier.
- `setvar_addr`: (optional) Specifies a symbol in the program image. This symbol will be rewritten with the base address of the I/O port.

The `setvar` element has the following attributes:

- `symbol`: Name of a symbol in the ELF file.
- `region_paddr`: Name of an MR. The symbol's value shall be updated to this MR's physical address. Note that on x86-64 platforms this MR must have a specified physical address. This restriction is due to x86-64 physical memory layout not being known at build-time.

The `protection_domain` element has the same attributes as any other protection domain as well as:

- `id`: The ID of the child for the parent to refer to.
- `setvar_id`: (optional) Specifies a symbol in the parent program image. This symbol will be rewritten with the ID of the child.

The `virtual_machine` element has the following attributes:

- `name`: A unique name for the virtual machine
- `priority`: The priority of the virtual machine (integer 0 to 254).
- `budget`: (optional) The VM's budget in microseconds; defaults to 1,000.
- `period`: (optional) The VM's period in microseconds; must not be smaller than the budget; defaults to the budget.

Additionally, it supports the following child elements:

- `vcpu`: (one or more) Describes the virtual CPU that will be tied to the virtual machine.
- `map`: (zero or more) Describes mapping of memory regions into the virtual machine.

The `vcpu` element has the following attributes:

- `id`: The vCPU identifier. Must be at least 0 and less than 62.
- `setvar_id`: (optional) Specifies a symbol in the program image. This symbol will be rewritten with the vCPU identifier.

The `map` element has the same attributes as the protection domain with the exception of `setvar_vaddr`.

7.2 `memory_region`

The `memory_region` element describes a memory region.

It supports the following attributes:

- `name`: A unique name for the memory region
- `size`: Size of the memory region in bytes (must be a multiple of the page size). This can be omitted if the memory region is has `prefill_path`, the size will be determined by the length of the file given.
- `page_size`: (optional) Size of the pages used in the memory region; must be a supported page size if provided. Defaults to the largest page size for the target architecture that the memory region is aligned to.
- `phys_addr`: (optional) The physical address for the start of the memory region (must be a multiple of the page size).
- `prefill_path`: (optional) Path to a file containing data that the memory region will be filled with at initialisation.

The `memory_region` element does not support any child elements.

7.2.1 Page sizes by architecture

Below are the available page sizes for each architecture that Microkit supports.

7.2.1.1 AArch64

- 0x1000 (4KiB)
- 0x200000 (2MiB)

7.2.1.2 RISC-V 64-bit

- 0x1000 (4KiB)
- 0x200000 (2MiB)

7.2.1.3 x86-64

- 0x1000 (4KiB)
- 0x200000 (2MiB)

7.3 `channel`

The `channel` element has exactly two `end` children elements for specifying the two PDs associated with the channel.

The `end` element has the following attributes:

- `pd`: Name of the protection domain for this end.
- `id`: Channel identifier in the context of the named protection domain. Must be at least 0 and less than 63.
- `pp`: (optional) Indicates that the protection domain for this end can perform a protected procedure call to the other end; defaults to false. Protected procedure calls can only be to PDs of strictly higher priority.
- `notify`: (optional) Indicates that the protection domain for this end can send a notification to the other end; defaults to true.
- `setvar_id`: (optional) Specifies a symbol in the program image. This symbol will be rewritten with the channel identifier.

The `id` is passed to the PD in the `notified` and `protected` entry points. The `id` should be passed to the `microkit_notify` and `microkit_ppcall` functions.

8 Board Support Packages

This chapter describes the board support packages that are available in the SDK.

The currently supported platforms are:

- ariane
- cheshire
- hifive_p550
- imx8mm_evk
- imx8mp_evk
- imx8mp_iotgate
- imx8mq_evk
- kria_k26
- maaxboard
- odroidc2
- odroidc4
- qemu_virt_aarch64
- qemu_virt_riscv64
- rockpro64
- rock3b
- rpi4b_1gb
- rpi4b_2gb
- rpi4b_4gb
- rpi4b_8gb
- serengeti
- star64
- tqma8xqp1gb
- ultra96v2
- x86_64_generic
- x86_64_generic_vtx
- zcu102

8.1 Ariane (CVA6)

Initial support is available for the CVA6 (formerly Ariane) core design on the Digilent Genesys2 board. CVA6 is an open-source RISC-V (rv64i) processor.

Microkit support expects that a compatible RISC-V SBI (e.g OpenSBI) has executed before jumping to the beginning of the loader image.

Microkit outputs an ELF for this platform.

You may compile OpenSBI with the Microkit image as a payload, or alternately install OpenSBI (with U-Boot optionally) to the SD card.

If you are booting from U-Boot, use the following command to start the system image:

```
=> bootelf ${image_addr}
```

Where `image_addr` is the load address of the ELF image.

Note that the OpenSBI version from the CVA6 SDK at the time of writing has issues when booting. It is recommended to use the mainline OpenSBI.

8.2 Compulab IOT-GATE-IMX8MPLUS

The IOT-GATE-IMX8PLUS is based on the NXP i.MX8MP SoC.

Microkit will produce a raw binary file by default, so when using U-Boot run the following command:

```
=> go 0x50000000
```

8.3 Cheshire

Support is available for Cheshire. It is an SoC design based on the CVA6 core, implementing a 64-bit RISC-V CPU.

Microkit outputs an ELF for this platform. Several steps are required in order to boot.

A custom version of OpenSBI is required. It can be found here. Build the firmware payload using platform `fpga/cheshire`.

To boot a Microkit image, use a GDB prompt via openOCD per seL4 Docs:

1. Reset board

   ```
   (gdb) monitor reset halt
   ```

2. Set the `a0` and `a1` registers for OpenSBI

   ```
   # tell OpenSBI where DTB is (there is none)
   (gdb) set $a0=0
   # tell OpenSBI that the default hart is #0
   (gdb) set $a1=0
   ```

3. Load OpenSBI's FW_JUMP payload targeted at `0x90000000`, implicitly setting the program counter.

   ```
   (gdb) load /path/to/opensbi/fw_jump.elf
   ```

4. Load the Microkit image:

   ```
   (gdb) restore /path/to/loader.img
   ```

5. Allow OpenSBI and Microkit to boot:

   ```
   (gdb) continue
   ```

8.4 i.MX8MM-EVK

Microkit will produce a raw binary file by default, so when using U-Boot run the following command:

```
=> go 0x41000000
```

8.5 i.MX8MP-EVK

Microkit will produce a raw binary file by default, so when using U-Boot run the following command:

```
=> go 0x41000000
```

8.6 i.MX8MQ-EVK

Microkit will produce a raw binary file by default, so when using U-Boot run the following command:

```
=> go 0x41000000
```

8.7 Kria K26

The Kria K26 is highly similar to the ZCU102, with slight differences in memory regions and the base UART controller.

To run the built image on the board, you have to use properly patched U-Boot - please see the section for ZCU102, for the details.

You have to load the binary file into memory and run it:

```
ZynqMP> tftpboot 0x40000000 loader.img
...
ZynqMP> go 0x40000000
```

A QEMU emulator is not available for the Kria K26 at this stage.

8.8 MaaXBoard

The MaaXBoard is a low-cost ARM SBC based on the NXP i.MX8MQ system-on-chip.

Microkit will produce a raw binary file by default, so when using U-Boot run the following command:

```
=> go 0x50000000
```

8.9 Odroid-C2

The HardKernel Odroid-C2 is an ARM SBC based on the Amlogic Meson S905 system-on-chip. It should be noted that the Odroid-C2 is no longer available for purchase but its successor, the Odroid-C4, is readily available at the time of writing.

Microkit will produce a raw binary file by default, so when using U-Boot run the following command:

```
=> go 0x20000000
```

8.10 Odroid-C4

The HardKernel Odroid-C4 is an ARM SBC based on the Amlogic Meson S905X3 system-on-chip.

Microkit will produce a raw binary file by default, so when using U-Boot run the following command:

```
=> go 0x20000000
```

8.11 Serengeti

Serengeti is a fork of the Cheshire platform by Trustworthy Systems. More information on the hardware platform can be found here.

Right now Serengeti is a superset of Cheshire by offering more peripherals than Cheshire has.

For running on Serengeti, please see Cheshire for instructions.

8.12 SiFive Premier P550

The SiFive Premier P550 is a development board based on the ESWIN EIC7700X system-on-chip.

Microkit will produce a uImage. To load the image you will need to use the `bootm` command. `bootm` expects an address of where the Device Tree Blob (DTB) will be as well as the image.

If you do not have a DTB for this platform you can obtain it from the seL4 source code with:

```
dtc -I dts -O dtb seL4/tools/dts/hifive-p550.dts > hifive-p550.dtb
```

Now you can load the images into U-Boot at the appropriate addresses, for example via TFTP:

```
=> setenv imgaddr "0xB0000000"
=> setenv dtbaddr "0xA0000000"
=> tftpboot ${imgaddr} /path/to/loader.img
=> tftpboot ${dtbaddr} /path/to/hifive-p550.dtb
=> bootm ${imgaddr} - ${dtbaddr}
```

`imgaddr` and `dtbaddr` are arbitrarily chosen addresses in main memory for this platform, you can load the image and DTB at another address.

8.13 QEMU virt (AArch64)

Support is available for the virtual AArch64 QEMU platform. This is a platform that is not based on any specific SoC or hardware platform and is intended for simulating systems for development or testing.

It should be noted that the platform support is configured with 2GB of main memory and the Cortex-A53 CPU. seL4 needs to know the memory layout and CPU at build-time so if you want to change these parameters (e.g to allow more memory), you will have to change the kernel options for `qemu_virt_aarch64` in `build_sdk.py`.

You can use the following command to simulate a Microkit system:

```
$ qemu-system-aarch64 \
    -machine virt,virtualization=on \
    -cpu cortex-a53 \
    -nographic \
    -serial mon:stdio \
    -device loader,file=[SYSTEM IMAGE],addr=0x70000000,cpu-num=0 \
    -m size=2G
```

When using the SMP configurations add the `-smp 4` argument to the QEMU command.

You can find more about the QEMU virt platform in the QEMU documentation.

8.14 QEMU virt (RISC-V 64-bit)

Support is available for the virtual RISC-V (64-bit) QEMU platform. This is a platform that is not based on any specific SoC or hardware platform and is intended for simulating systems for development or testing.

It should be noted that the platform support is configured with 2GB of main memory. seL4 has a static amount of memory at build-time, if you want allow for more memory, you will have to change the kernel options for `qemu_virt_riscv64` in `build_sdk.py`.

You can use the following command to simulate a Microkit system:

```
$ qemu-system-riscv64 \
    -machine virt \
    -nographic \
    -serial mon:stdio \
    -kernel [SYSTEM IMAGE] \
    -m size=2G
```

When using the SMP configurations add the `-smp 4` argument to the QEMU command.

QEMU will start the system image using its packaged version of OpenSBI.

You can find more about the QEMU virt platform in the QEMU documentation.

8.15 Raspberry Pi 4B

Support is available for the Raspberry Pi 4 Model B. There are multiple models of the Raspberry Pi 4B that have different amounts of RAM, we have support for the 1GB, 2GB, 4GB and 8GB models. Because the amount of RAM must be known statically by seL4, the Microkit board names differ for each model:

- `rpi4b_1gb` for 1GB of RAM.
- `rpi4b_2gb` for 2GB of RAM.
- `rpi4b_4gb` for 4GB of RAM.
- `rpi4b_8gb` for 8GB of RAM.

For initial board setup, please see the instructions on the seL4 website.

When getting into the U-Boot console you want to load the Microkit binary image to address 0x10000000 and then run `go 0x10000000`.

For example, if you were to load the image via the MMC you would run the following U-Boot commands:

```
=> fatload mmc 0 0x10000000 <SYSTEM IMAGE>
=> go 0x10000000
```

8.16 Pine64 ROCKPro64

Microkit will produce a raw binary file by default, so when using U-Boot run the following command:

```
=> go 0x30000000
```

8.17 Radxa ROCK 3B

Support is available for the Radxa ROCK 3B platform which is based on the Rockchip RK3568 SoC.

Since the platform relies on some closed-source binary blobs for first stage bootloader and then ARM's TrustZone A, we need to compile the U-Boot including these images. Detailed instructions on how to do that are available here.

Once the proper U-Boot image is in place, you can simply load the `loader.img` on the board and run it like that (this is assuming you have the TFTP server set up):

```
=> tftp 0x30000000 loader.img
=> go 0x30000000
```

For more booting options, please refer to the seL4 board setup guide.

8.18 Pine64 Star64

Support is available for the Pine64 Star64 platform which is based on the StarFive JH7110 SoC.

The platform has a 4GB and 8GB model, we assume the 4GB model.

The default boot flow of the Star64 is: 1. OpenSBI 2. U-Boot 3. Operating System

This means that the system image that Microkit produces does not need to be explicitly packaged with an SBI implementation such as OpenSBI.

Microkit will produce a uImage. To load the image you will need to use the `bootm` command. `bootm` expects an address of where the Device Tree Blob (DTB) will be as well as the image.

If you do not have a DTB for this platform you can obtain it from the seL4 source code with:

```
dtc -I dts -O dtb seL4/tools/dts/star64.dts > star64.dtb
```

Now you can load the images into U-Boot at the appropriate addresses, for example via TFTP:

```
=> setenv imgaddr "0xE0000000"
=> setenv dtbaddr "0xD0000000"
=> tftpboot ${imgaddr} /path/to/loader.img
=> tftpboot ${dtbaddr} /path/to/star64.dtb
=> bootm ${imgaddr} - ${dtbaddr}
```

`imgaddr` and `dtbaddr` are arbitrarily chosen addresses in main memory for this platform, you can load the image and DTB at another address.

8.19 TQMa8XQP 1GB

The TQMa8XQP is a system-on-module designed by TQ-Systems GmbH. The modules incorporates an NXP i.MX8X Quad Plus system-on-chip and 1GiB ECC memory.

TQ-Systems provide the MBa8Xx carrier board for development purposes. The instructions provided assume the use of the MBa8Xx carrier board. If you are using a different carrier board please refer to the appropriate documentation.

Note: There are different configured of the TQMa8Xx board which include different NXP SoCs and different memory configurations. Such modules are not supported.

The MBa8Xx provides access to the TQMa8XQP UART via UART-USB bridge. To access the UART connect a USB micro cable to port **X13**. The UART-USB bridge supports 4 individual UARTs; the UART is connected to the 2nd port.

By default the SoM will autoboot using U-Boot. Hit any key during the boot process to stop the autoboot.

A new board will autoboot to Linux. You will likely want to disable autoboot:

```
=> env set bootdelay -1
=> env save
```

The board can be reset by pressing switch **S4** (located next to the Ethernet port). Alternatively, you can use the `reset` command from the U-Boot prompt.

During development the most convenient way to boot a Microkit image is via network booting. U-Boot support booting via the *tftp* protocol. To support this you'll want to configure the network. U-Boot supports DHCP, however it is often more reliable to explicitly set an IP address. For example:

```
=> env set ipaddr 10.1.1.2
=> env set netmask 255.255.255.0
=> env set serverip 10.1.1.1
=> env save
```

To use tftp you also need to set the file to load and the memory address to load it to:

```
=> env set bootfile loader.img
=> env set loadaddr 0x80280000
=> env save
```

The system image generated by the Microkit tool by default is a raw binary file.

An example sequence of commands for booting is:

```
=> tftpboot
=> dcache flush
=> icache flush
=> go ${loadaddr}
```

Rather than typing these each time you can create a U-Boot script:

```
=> env set microkit 'tftpboot; dcache flush; icache flush; go ${loadaddr}'
=> env save
=> run microkit
```

When debugging is enabled the kernel will use the same UART as U-Boot.

8.20 Ultra96V2

To run the built image on the board, you have to use properly patched U-Boot - please see the section for ZCU102, for the details.

You have to load the binary file into memory and run it:

```
ZynqMP> tftpboot 0x40000000 loader.img
...
ZynqMP> go 0x40000000
```

8.21 x86-64 generic

This board supports x86-64 platforms with generic microarchitecture and no virtualisation.

Up to 16 CPU cores are supported. If your platform's CPU count exceeds the maximum, the excess CPUs are ignored by seL4. This limitation is due to a build-time config option of seL4, to increase the limit, see the board configuration in `build_sdk.py`.

On x86-64, Microkit produces 3 ELF images in the output directory:

- `sel4.elf`, seL4 kernel (64-bit ELF)
- `sel4_32.elf`, seL4 kernel (32-bit ELF)
- Initial Task image (Multiboot Boot Module ELF)

The filename of the initial task image is specified with the `-o` option, and defaults to loader.img. Both kernel ELF files are written to the same directory as the initial task image.

Although this target is for x86-64, your bootloader may require a 32-bit kernel ELF.

When using QEMU, you must use the 32-bit kernel ELF, for example:

```
$ qemu-system-x86_64 \
    -cpu qemu64,+fsgsbase,+pdpe1gb,+xsaveopt,+xsave \
    -m "1G" \
    -display none \
    -serial mon:stdio \
    -kernel sel4_32.elf \
    -initrd loader.img
```

Unlike Microkit's `qemu_virt_aarch64` and `qemu_virt_riscv64`, the amount of memory provided to QEMU is configurable at run time with the `-m` option, rather than at build time.

QEMU currently does not support 64-bit kernel ELFs and gives the following error:

```
qemu-system-x86_64: Cannot load x86-64 image, give a 32bit one.
```

To obtain a bootable ISO image, a Multiboot 2-compliant bootloader must be used. Please refer to your bootloader's documentations.

8.22 x86-64 generic (with VT-x)

This board supports x86-64 platforms with generic microarchitecture and virtualisation.

This configuration assumes Intel VT-x support. AMD-V is (at this time) not supported by seL4, so running x86-64 virtual machines on Microkit will require an Intel CPU with VT-x.

Currently, only 1 vCPU is supported per virtual machine on x86-64.

The boot process is identical to x86-64 generic.

This configuration exists because some x86 emulators (for example, QEMU on macOS on Apple Silicon) do not emulate Intel VT-x.

If you see the following message at boot:

```
vt-x: not supported
```

Consider using the non VT-x configuration: x86-64 generic, or switch to a x86 emulator that emulates Intel VT-x, such as Bochs.

8.23 ZCU102

The ZCU102 can run on a physical board or on an appropriate QEMU based emulator.

Microkit will produce a raw binary file by default, so when using U-Boot run the following command:

```
=> go 0x40000000
```

For simulating the ZCU102 using QEMU, use the following command:

```
$ qemu-system-aarch64 \
   -m size=4G  \
   -machine xlnx-zcu102,virtualization=on \
   -nographic \
   -device loader,file=[SYSTEM IMAGE],addr=0x40000000,cpu-num=0 \
   -serial mon:stdio
```

When using the SMP configurations add the `-smp 4` argument to the QEMU command.

It should be noted that when using U-Boot to load and run a Microkit system image, that there may be additional setup needed.

For the ZynqMP class of platforms, which the ZCU102 is apart of, U-Boot does not start the Microkit system Exception Level 2 (EL2) which is necessary for Microkit to start (this is because seL4 is configured as a hypervisor).

You can see that when using the `go` command, U-Boot is unconditionally always dropping down to EL1.

To avoid this behaviour, the call to `armv8_switch_to_el1` should be replaced with `armv8_switch_to_el2` in this `do_go_exec` function.

8.24 Adding Platform Support

The following section is a guide for adding support for a new ARM or RISC-V platform to Microkit.

8.24.1 Prerequisites

Before you can start with adding platform support to Microkit, the platform must be supported by the seL4 kernel. You can find information on how to do so here.

8.24.2 Getting Microkit Components Working

8.24.2.1 build_sdk.py The first step to adding Microkit support is to modify the `build_sdk.py` script in order to build the required artefacts for the new platform. This involves adding to the `SUPPORTED_BOARDS` list with the `BoardInfo` options containing the platform specific attributes. This should be fairly self-explanatory by looking at the existing entries with the exception of the `loader_link_address`.

The `loader_link_address` parameter specifies the physical address of where the bootloader for Microkit (which is responsible for setting up the system before seL4 starts) is going to be loaded. This address needs to match where in main memory the final system image is actually loaded (e.g where a previous bootloader such as U-Boot loads the image to). This means that the address is restricted to the platform's main memory region.

8.24.2.2 Loader

8.24.2.2.1 UART The other component of Microkit that is platform dependent is the loader itself. The loader will attempt to access the UART for debug output which requires a basic `putc` implementation. The UART device used in the loader should be the same as what is used for the seL4 kernel debug output.

It should be noted that on RISC-V platforms, the SBI will be used for `putc` so no porting is necessary.

8.24.2.2.2 CPU configuration For supporting multi-core systems on Microkit, the loader needs to know the platform-specific CPU identifiers that we want to make use of.

For ARM platforms, a `psci_target_cpus` array needs to be defined which contains the PSCI `target_cpu` value for each CPU. This value can be found by looking at the `reg` field for each CPU in the Device Tree.

For RISC-V platforms, a `hart_ids` array needs to be defined which contains the Hart ID for each CPU. This value can be found by looking at the `reg` field for each CPU in the Device Tree. The first value must be the same as `FIRST_HART_ID` defined in seL4.

8.24.2.3 Next steps

Once you have patched the loader and the SDK build script, there should be no other changes required to have a working platform port. It is a good idea at this point to boot a hello world system to confirm the port is working.

If there are issues with porting the platform, please open an issue on GitHub.

8.24.3 Contributing Platform Support

Once you believe that the port works, you can open a pull request with required changes as well as documentation in the manual about the platform and how to run Microkit images on it.

Please also update the `platforms.yml` file, you can find more information at the top of the file.

9 Rationale

This section describes the rationales driving the Microkit design choices.

9.1 Overview

The seL4 microkernel provides a set of powerful and flexible mechanisms that can be used for building almost arbitrary systems. While minimising constraints on the nature of system designs and scope of deployments, this flexibility makes it challenging to design the best system for a particular use case, requiring extensive seL4 experience from developers.

The Microkit addresses this challenge by constraining the system architecture to one that provides enough features and power for its target usage class (IoT, cyberphysical and other embedded systems with a static architecture), enabling a much simpler set of developer-visible abstractions.

9.2 Protection Domains

PDs are single-threaded to keep the programming model and implementations simple, and because this serves the needs of most present use cases in the target domains. Extending the model to multithreaded applications (clients) is straightforward and can be done if needed. Extending to multithreaded services is possible but requires additional infrastructure for which we see no need in the near future.

9.3 Protected Procedure Priorities

The restriction of only calling to higher priority prevents deadlocks and reflects the notion that the callee operates on behalf of the caller, and it should not be possible to preempt execution of the callee unless the caller could be preempted as well.

This greatly simplifies reasoning about real-time properties in the system; in particular, it means that PPs can be used to implement *resource servers*, where shared resources are encapsulated in a component that ensures mutual exclusion, while avoiding unbounded priority inversions through the *immediate priority ceiling protocol*.

While it would be possible to achieve the same by allowing PPs between PDs of the same priority, this would be much harder to statically analyse for loop-freedom (and thus deadlock-freedom). The drawback is that we waste a part of the priority space where a logical entity is split into multiple PDs, eg to separate out a particularly critical component to formally verify it, when the complete entity would be too complex for formal verification. For the kinds of systems targeted by the Microkit, this reduction of the usable priority space is unlikely to cause problems.

9.4 Protected Procedure Argument Size

The limitation on the size of by-value arguments is forced by the (architecture-dependent) limits on the payload size of the underlying seL4 operations, as well as by efficiency considerations. The protected procedure payload should be considered as analogous to function arguments in the C language; similar limitations exist in the C ABIs (Application Binary Interfaces) of various platforms.

9.5 Limits

The limitation on the number of protection domains in the system is relatively arbitrary. Based on experience with the system and the types of systems being built it is possible for this to be increased in the future.

The limitation on the number of channels for a protection domain is based on the size of the notification word in seL4. Changing this to be larger than 64 would most likely require changes to seL4. The reason for why the limit is not a power of two is due to part of the notification word being for internal libmicrokit use.

10 Internals

The following section describes internal details for how the Microkit works and all the components of Microkit. As a user of Microkit, it is not necessary know this information, however, there is no harm in having a greater understanding of the tools that you are using.

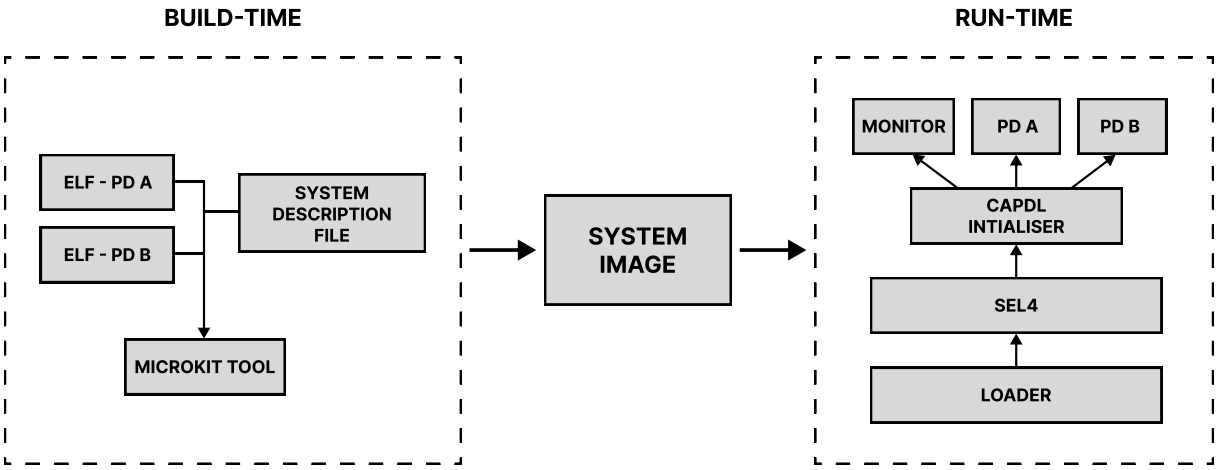

Figure 1: Microkit flow

The diagram above aims to show the general flow of a Microkit system from build-time to run-time.

The user provides the SDF (System Description File) and the ELFs that correspond to PD program images to the Microkit tool which is responsible for packaging everything together into a single image for the target platform.

10.0.1 ARM and RISC-V

The final image is a bootable image which contains a couple different things:

- the Microkit loader
- seL4
- the capDL initialiser (and associated capDL specification)

When booting the image, the Microkit loader starts, jumps to the kernel, which starts the capDL initialiser, which then sets up the entire system and starts all the PDs, including the Monitor.

10.0.2 x86

The tool produces the kernel image and boot module image that contains the capDL initialiser. You must bring your own Multiboot 2 compliant bootloader which then loads the kernel and boot module.

Now, we will go into a bit more detail about each of these stages of the booting process as well as what exactly the Microkit tool is doing.

10.1 Loader (ARM and RISC-V)

The loader starts first, it has two main jobs:

1. Unpack all the parts of the system (kernel and initialiser) into their expected locations within main memory.
2. Finish initialising the hardware such that the rest of the system can start.

Before the Microkit loader starts, there would most likely have been some other bootloader such as U-Boot or firmware on the target that did its own hardware initialisation before starting Microkit.

However, there are certain things that seL4 expects to be initialised that will not be done by a previous booting stage, such as:

- changing to the right exception level
- enabling the MMU (seL4 expects the MMU to be on when it starts)
- interrupt controller setup

Once this is all completed, the loader jumps to seL4 which starts executing. The loader will never be executed again.

10.2 capDL Initialiser

Once the kernel has done its own initialisation, it will begin the 'initial task'. On seL4, this is a thread that contains all the initial capabilities to resources and free memory that are used to setup the rest of the system.

Within a Microkit environment, the 'initial task' is the capDL initialiser. Its main job is to initialises and starts the system in conformance with the capDL specification.

At build-time, the Microkit tool embeds the capDL specification that describe all kernel objects that needs to be created. Then for each kernel object, the spec describe what state they need to be in and what capabilities exist to that object (i.e. who has access to this kernel object). For example, the spec would specify the: - starting Instruction Pointer (IP), Stack Pointer (SP) and IPC buffer pointer of a Thread Control Block (TCB), - page table structure and mapping attributes of an address space (VSpace), - interrupts (IRQ), - scheduling parameters (Scheduling Context), - and so on...

10.3 Monitor

The Monitor is a special PD that executes at the highest priority. It's job is to receive any faults caused by protection domains crashing or causing exceptions.

After the monitor initialises itself, goes to sleep and waits for any faults from protection domains. On debug mode, this results in a message about which PD caused an exception and details on the PD's state at the time of the fault.

Other than printing fault details, the monitor does not do anything to handle the fault, it will simply go back to sleep waiting for any other faults.

10.4 libmicrokit

Unlike the previous sections, libmicrokit is not its own program but it is worth a mention since it makes up the core of each protection domain in a system.

When each PD starts, we enter libmicrokit's starting point which does some initial setup and calls the `init` entry point specified by the user. Once that completes, the PD will enter libmicrokit's event handler which sleeps until it receives events.

These events could be notifies (from other PDs or from an interrupt), PPCs, and faults. For each event, the appropriate user-specified entry point is called and then when it returns the PD goes back to sleep, waiting on any more events.

10.5 Microkit tool

The Microkit tool's ultimate job is to take in the description of the user's system, the SDF, and convert into an seL4 system that boots and executes.

There are obvious steps such as parsing the SDF and PD ELFs. But the majority of the work done by the tool is converting the system description into the capDL specification then embedding it into the capDL initialiser image.

10.5.1 ARM & RISC-V specific

Since the physical memory layout of ARM and RISC-V platforms can be determined at boot time by reading the device tree. The Microkit tool goes one step further and statically check that the produced capDL specification is bootable on the target board. For example, we check that there are no frames outside of device or normal memory, and that there are enough memory to create all required kernel objects.

In order to do this however, the Microkit tool needs to emulate how the seL4 kernel boot to obtain the list of free untyped objects that the kernel would give to the initial task.

While this is non-trivial to do, it comes with the useful property that if the tool produces a valid image, there should be no errors upon initialising the system If there are any errors with configuring the system (e.g running out of memory), they will be caught at build-time. This can only reasonably be done due to the static-architecture of Microkit systems.