

Muen System Specification

Adrian-Ken Rueegsegger, Reto Buerki

v0.7.1, February 8, 2023

無 縄

Copyright © 2023 codelabs GmbH
Copyright © 2023 secunet Security Networks AG

Further publications, reprints, duplications or recordings - no matter in which form, of the entire document or parts of it - are only permissible with the prior consent of codelabs GmbH or secunet Security Networks AG.

Contents

1	Introduction	5
2	System Policy	6
2.1	Policy Format	6
3	System Integration	8
4	Tau0 Concept	11
4.1	Static Mode of Operation	11
4.2	Dynamic	11
5	Toolchain	13
5.1	Overview	13
5.2	Core Tools	16
5.3	Additional Tools	23
6	Policy Validation	25
6.1	Configuration	25
6.2	Devices	25
6.3	Device Domains	26
6.4	Events	26
6.5	Files	27
6.6	Hardware	27
6.7	Kernel	28
6.8	Memory	28
6.9	Model Specific Registers (MSR)	30
6.10	Platform	30
6.11	Scheduling	30
6.12	Subjects	31
7	Policy Structure	33
7.1	Policy Schema Documentation	33
8	Appendix	97
8.1	Annotated Example Policy	97
9	Bibliography	133

List of Figures

3.1 System Integration	9
5.1 Build process	14

Listings

5.1	τ_0 Command Stream	19
8.1	Demo System (VT-d)	97

Chapter 1

Introduction

The Muen system policy is a description of a component-based system running on top of the Muen Separation Kernel (SK). It defines what hardware resources are present, how many active components (called subjects) the system is composed of, how they interact and which system resources they are allowed to access. The contents a Muen system policy is composed of are outlined in chapter 2.

A system integrator specifies and configures such a component-based system at integration time in XML format. The Muen toolchain transforms the system description in multiple steps to the final system description, resolving abstractions which exist to make life simpler and less error-prone to the integrator. Additionally, the toolchain also creates various build artifacts which are incorporated into the system image. Chapter 3 gives an overview of the system integration process.

The Muen SK can be regarded as a policy enforcement engine, in the sense that it has no knowledge about the actual content of the generated data structures and in consequence the policy. For example, it knows nothing about the contents of subject page tables which define a subject's address space, nor does it know anything about its own page tables. In fact, these structures are not even mapped into the kernel.

The most important and final step in the integration of a Muen system is the actual generation of the data structures which guarantee subject isolation and the composition of the final system image. This step is performed by a trusted system composer called (static) τ_0 (Tau Zero). The concept of τ_0 is introduced in chapter 4.

Section 5 explains every tool and the system image composer in detail. It also presents the usage of each tool. Section 6 then outlines all semantic checks performed on the system policy primarily by the validation tool, but also by other tools in the toolchain.

Finally section 7 specifies the XML schema and structure of the source format of the Muen system policy. Explanations and examples illustrate how to configure a component-based system with the Muen SK.

Chapter 2

System Policy

The Muen policy specifies the following properties of a system:

- Configuration values
- Hardware resources
- Platform description
- Physical memory regions
- Device domains
- Events
- Communication channels
- Components
- Subjects
- Scheduling plans

The policy serves as a static description of a Muen system. Since all aspects of the system are fixed at integration time the policy is very well suited for automated as well as manual validation prior to system execution.

The details of each property above is outlined with examples in the XSD-schema of the format source policy in section 7.

2.1 Policy Format

The system policy is specified in XML. There are currently three different policy formats:

- Source Format
- Format A
- Format B

The motivation to have several policy formats is to provide abstractions and a compact way for users to specify a system in format source while simultaneously facilitate traceability as well as reduced complexity of tools operating on the policy formats A and B.

The implementation of such tools is simplified by the absence of higher-level abstractions in the latter formats which would make the extraction of input data more involved.

Furthermore, the final format B must specify every aspect of the system explicitly, e.g. all attributes have a concrete value assigned, something which would be very tedious and repetitive and that burden should not be put on an integrator.

The following sections give more detail about each policy format.

 Only the policy in format source intended for system integrators is specified in this document. Other formats are processed by the toolchain and thus considered *internal*. While it is possible to specify a system policy in format A or B, it is not recommended.

2.1.1 Source Format

The user-specified policy is written in the so called *source format*. Constructs such as channels provide abstractions to simplify the specification of component-based systems. Many XML elements and attributes are optional and are *expanded* during later steps of the policy compilation process.

Kernel and τ_0 subject (4.2) resources are not part of the source format since they are automatically added as part of the policy expansion step.

The use of configuration values enables parametrization of the system policy.

The policy in source format is specified in detail in section 7, while appendix 8.1 provides an annotated example policy illustrating the various policy elements.

2.1.2 Format A

Format A is a processed version of the source format where all inclusions of external files are resolved and abstractions such as channels have been deconstructed into their constituent parts. For example, a channel is expanded to a physical memory region and the corresponding writer and reader subject mappings with the appropriate access rights. Optional associated events have been automatically created and correctly linked with the designated subjects.

In this format all implicit elements, such as for example automatically generated page table memory regions, are specified. The kernel and τ_0 configuration is also declared as part of format A.

The only optional attributes are addresses of physical memory regions.

2.1.3 Format B

Format B is equivalent to Format A except that all physical memory regions have a fixed location (i.e. their physical address is set).

Chapter 3

System Integration

A Muen system defined via the system policy is transformed and integrated by various tools to generate a bootable system image.

The directed graph 3.1 on page 9 illustrates the process.

At the top, the graph shows how configuration and build parameters are applied to the following constituents of the system policy:

- Hardware description (static)
Contains manually specified devices by the integrator, e.g. common hardware like I/O ports of a PC speaker. Such devices are not automatically collected by the hardware configuration generator.
- Hardware description (generated)
Hardware description extracted from a running Linux system by the mugenhwcfg tool (section 5.3.3).
- Platform description
Common names and abstractions to form a unifying view over different hardware configurations. Additionally, platform-specific configuration values can be provided here.
- System description
Specification of an actual component-based system running on the Muen SK.

These combined inputs form the parameterized system policy in format source, which can be used by components to extract system information. Such information might be for example the log channel count of a debug server subject, or whether a specific debug facility has been enabled by the system integrator.

The CSPECs mechanism outlined in the *Muen Component Specification* document [1] can be used by components to generate source specifications (e.g. in SPARK/Ada) from the component description. Furthermore, a component might expand its own component description with information extracted from system information, or it might use the mucbinsplit tool (section 5.3.4) to automatically fill in the memory regions provided by its binary after compilation. The expanded component description is then merged with the system policy for further processing.

After all component descriptions have been merged into the system policy, it is expanded by the expander tool (section 5.2.3). This step transforms the system policy from format source to format A. Abstractions like directed channels are now resolved to basic shared memory mechanisms and events, non-present optional attributes are added and set to default values.

The allocator tool (5.2.4) then loops over all physical memory regions which have no address assigned and places them in memory by allocating a region and thus a physical start address from the usable pool. The usable pool information is extracted from the allocatable memory block list (7.1.17) in the system policy. This process transforms the policy to format B where all elements must be present and attributes specified.

The policy is then checked for consistency and configuration errors by the validator tool (5.2.5). If a misconfiguration is found, the user is informed and the build aborts. The extensive checks performed by the validator tool are listed in section 6. If no error is found, the system policy is then ready to be used for three subsequent steps:

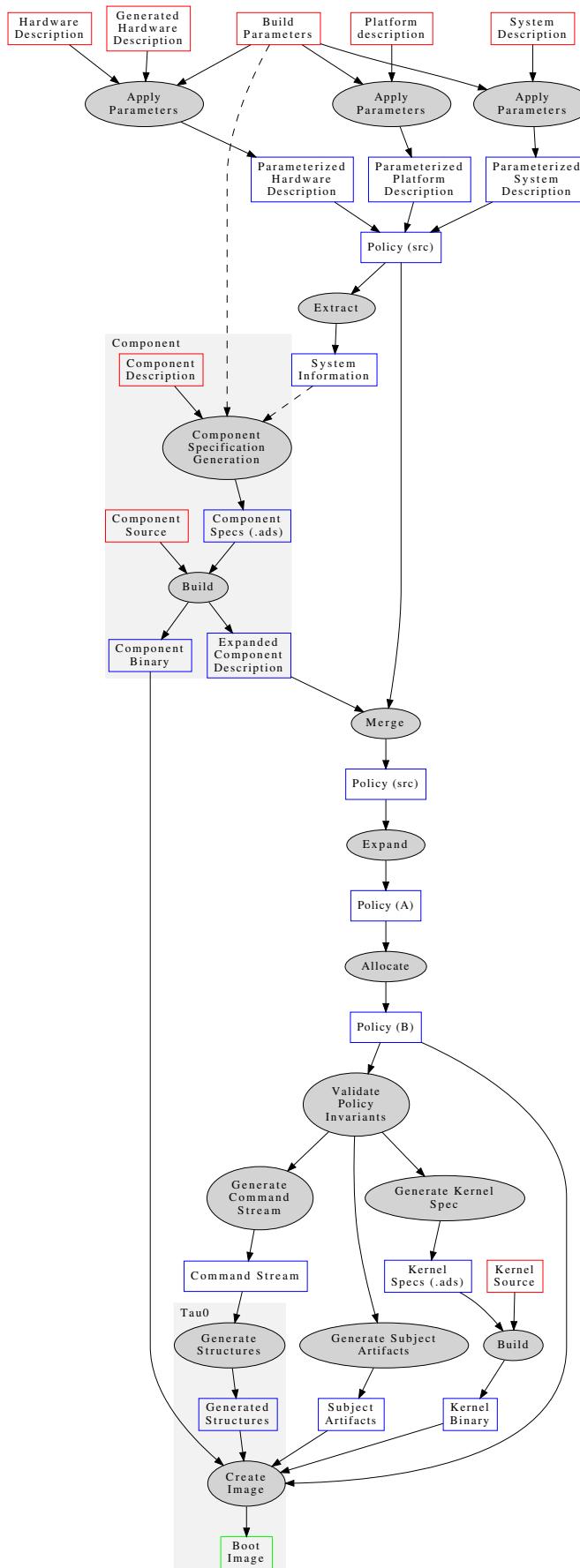


Figure 3.1: System Integration

- Generate kernel specifications (SPARK/Ada source files)
- Generate structures for subjects
- Create a command stream for τ_0

The kernel source specifications contain tables and constants which represent the policy that is compiled into the kernel as part of the kernel build process and enforced at runtime.

An example of generated subject structures are ACPI tables, which are mapped into a Linux VM to announce the available hardware resources.

The command stream generator ([5.2.7](#)) generates instructions in XML format for the τ_0 system composer explained in the following section.

Chapter 4

Tau0 Concept

The XML command stream together with the other build artifacts like subject structures or the kernel binary is provided as input to the trusted system composer τ_0 . Its task is to compose a system image while making sure that certain invariants are not violated. The τ_0 concept is a mechanism to gradually increase the flexibility of a component based system while keeping a high level of assurance regarding the correctness of isolation enforcement.

There are two modes of operation for τ_0 :

- static
- dynamic

In the static scenario, the task of τ_0 is to construct a bootable system image by assembling the input files and generating data structures such as page tables, all while checking that invariants necessary for correct isolation are valid. An example for such an invariant is that no subject memory mapping may reference a memory region containing paging structures.

For the dynamic case, the goal is to have a trusted τ_0 subject with additional privileges to interact with the Muen SK over a special τ_0 interface. This will allow τ_0 to change certain clearly defined aspects of the system state at runtime. A potential use-case would be to set up a new subject, assign resources like memory and devices to it and then instruct the kernel to schedule it.

Since it must be guaranteed that a dynamic system is as secure as the static one, τ_0 must be developed with the same care as the kernel itself, meaning it must be written in SPARK/Ada and security properties which provide hard isolation must be formally proven. This process is currently ongoing.

4.1 Static Mode of Operation

The static variant is the one which is currently implemented. τ_0 runs at integration time and assembles the system image by constructing the data structures guaranteeing isolation and merging in the build artifacts of the other Muen tools, like untrusted ACPI data structures for Linux VMs.

Static τ_0 fulfills its task by creating the system image in memory while processing the commands from the command stream. See listing 5.1 on page 19 for an example command stream.

τ_0 is written in SPARK/Ada and it applies memory typization to formally prove aspects of the system. Command processing starts from a well-known good state and it is enforced that each system state transition resulting from a new command input results in a good state again by showing that invariants hold after the transition. If not, the command is rejected and the build aborts.

See the project README or the webpage¹ for more information about the current state of τ_0 .

4.2 Dynamic

While the system image is composed by the static variant of τ_0 , the goal is to run the same code as τ_0 subject at runtime. Note that this is not yet implemented but planned as a way

¹<https://muen.sk/tau0.html>

forward to achieve more dynamic systems while having the same assurance about security and safety properties.

The dynamic τ_0 running as subject will reconstruct the system state defined at integration time and continue to process commands starting from there. Depending on the system use case, commands might be sent to dynamic τ_0 by a special control subject.

The dynamic variant can be divided into multiple sub-variants, depending on how much dynamic system behavior is allowed. For example, the initial dynamic variant might only allow entity construction, not destruction.

Chapter 5

Toolchain

5.1 Overview

While the previous section 3 presented an overview of the system integration process and section 4 introduced the τ_0 concept, this section focuses on the detailed description of the tools forming the Muen toolchain.

The tool-based processing of the Muen system policy can be divided into the following steps:

- Policy merging
- Components build
- Components specification merging
- Policy compilation
- Policy validation
- Structure generation
- Command stream generation for τ_0
- Image generation by τ_0

Following the Unix philosophy "A program should do only one thing and do it well" each of the tools only performs a specific task. They work in conjunction to process a user-defined policy and build a bootable system image. Figure 5.1 presents another illustration of the policy processing, this time laying the focus on the tools. The following sections explain each processing step while section 5.2 describes each tool separately.

5.1.1 Policy Merging

The Merger tool outlined in section 5.2.1 is responsible to merge XML files stored at different locations on the file system into one system policy in format source.

The tool reads a system configuration in XML format to locate the following files:

- System policy
- Hardware specification
- Additional hardware specification
- Platform specification

To make the system description flexible and modular the following features are supported in the input policy:

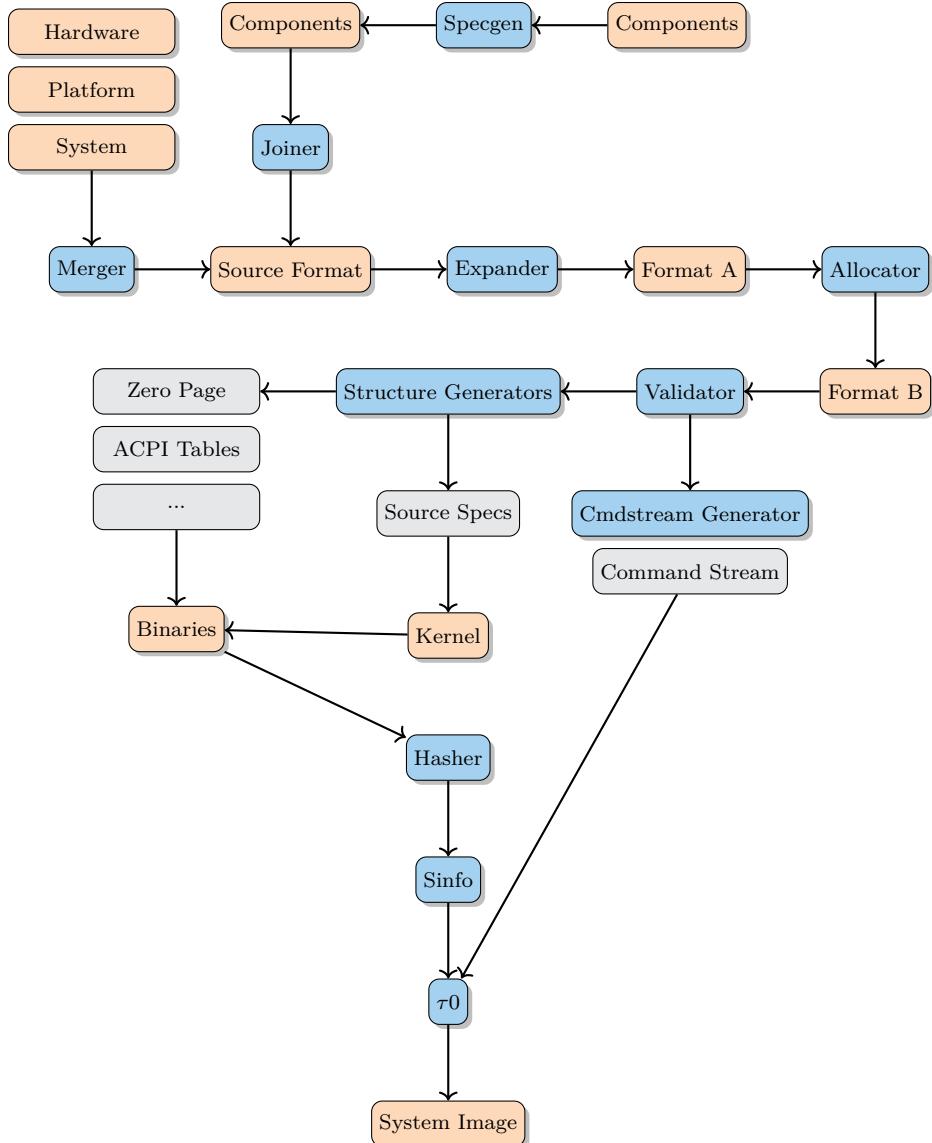


Figure 5.1: Build process

- The tool provides an implementation of the XML XInclude mechanism¹. Using includes, the policy writer is able to separate and organize the system policy as desired. Instead of specifying the whole policy in one file, the subject specifications can be split into separate files, or common parts shared by different system descriptions can be extracted.
- Expressions can be used to formulate (nested) terms using equality/inequality, numeric and logical operators as well as concatenation of strings. Expressions can be used just like configuration variables to provide parameters for other mechanisms.
- The use of conditionals enables selective activation of parts of the source policy depending on the value of a given configuration variable. This allows flexible customization of a system during policy compilation by setting the value of a configuration variable or formulating an appropriate expression.
- Configuration variable substitution enables the policy writer to set the value of attributes to those of referenced configuration variables or expressions. Attributes that start with a dollar sign followed by a variable name are substituted by the value of the variable.

¹<http://www.w3.org/TR/xinclude-11/>

- The policy may define templates for XML code, including parameters that can be used within expressions, conditionals and references within that template. Templates can be employed to avoid code duplication and to encapsulate portions of code. Hence, templates can define building blocks and help to provide a high-level view of a system.
- When two building blocks are connected via a channel or when a subject behaves like a client of another subject, it is desirable to insert a communication channel into a subject from “outside” of that subject. Such additions to an XML node are possible with amend statements. On evaluation, the children of an amend-node are merged into the children of the node specified by the given XPath.

After the merge step, the resulting policy is well formatted to minimize the difference in the generated policies resulting from the subsequent tasks. This allows the user to easily review (`diff`) and therefore verify the results of each policy compilation task.

5.1.2 Components Build

After hardware, platform and high-level system policy are merged into a single source policy file, components may extract relevant information. For example an XSL transformation (XSLT) script could extract the I/O port of a specific device and create a corresponding configuration value based on it, which is then included in the component specification.

The `mucgenspec` tool described in [5.2.9](#) implements the blue *Specgen* task shown in figure [5.1](#). It is used to process component specifications and, similar to the policy merger, supports conditionals, expressions and configuration value substitutions. It also generates Ada/SPARK packages containing constants derived from the declared component resources and config values. These constants can be used to reliably address specific or configurable resources in the source code.

After the component specification has been processed, the component source code is compiled into a binary.

The `mucbinsplit` tool described in [5.3.4](#) can be used to extract ELF sections of the component binary into separate files. It automatically extends the component specification by adding a corresponding memory region with the appropriate access rights (e.g. executable, writable) for text, rodata, data, bss and stack sections.

5.1.3 Components Specification Merging

The processed component specifications are merged into the system source policy by the Muen component specification joiner tool described in section [5.2.2](#).

This step is optional as static component specifications which need no processing can also be manually specified in the system policy directly.

5.1.4 Policy Compilation

Policy compilation encompasses the tasks involved to transform the policy from source format to format A and finally to format B, which is the fully expanded format with no implicit properties.

The Expander tool takes care of completing the user-specified policy with additional information and resolving abstractions only available in format source to their corresponding low-level constructs.

For example, the concept of *channels* only exists in format source. Therefore a channel specified in format source must be expanded to shared memory regions with optional associated events in format A. Also, the Expander tool inserts specifications for the Muen kernel itself so the user is lifted from that burden. Generally, the aim of the expansion task is to make the life of a policy writer as easy as possible by expanding all information which can be derived automatically. Section [5.2.3](#) explains the Expander tool in detail.

The result of the expansion task is a policy in format A which is the input for the Allocator tool. This tool is responsible to assign physical memory addresses to all memory regions which are not already explicitly placed in memory. By querying the hardware section of the policy, the tool is aware of the total amount of available RAM on a specific system and allocates regions of it for memory elements with no explicit physical address. The Allocator tool also implements

optimization strategies to keep the resulting system image as small as possible. For example, file-backed memory regions (e.g. a memory region storing a component executable) are preferably placed in lower physical regions. See section 5.2.4 for a description of the Allocator tool.

After the allocation task is complete, the policy is stored in format B. This format states all system properties explicitly and is used as input for the Validation step.

5.1.5 Policy Validation

Before structures required to pack the final system image are generated, the policy must be thoroughly validated to catch errors in the system specification. Such errors might range from overlapping memory, undefined resource references to incomplete scheduling plans etc. The Validator task performs checks that assure the policy in format B is sound and free from higher-level errors that are not covered by XML schemata restrictions.

It is important to always run the Validator as the system could otherwise exhibit unexpected behavior. This is especially true if a policy writer decides to specify the system directly in format B which is also possible but not advised. Section 5.2.5 explains the usage of the Validator tool, while section 6 outlines all performed checks.

It should be noted that correct memory typization and all invariants enforced by τ_0 when constructing the system image cannot be bypassed, since the checks are inherent to the generation of the bootable image file.

5.1.6 Structure Generation

The structure generation step encompasses various tools which extract information from a policy in format B and generate files in different formats.

While some generated files are directly linked into the Muen kernel (i.e. Source Specifications, see 5.2.9), most of them are subject-related. Depending on the subjects included in the actual system policy, the following subject structures are generated:

- MSR store regions
- Sinfo regions
- Regions for Linux VMs
 - ACPI tables
 - Linux zero-page (ZP) regions
- Regions for MirageOS/Solo5² unikernels
 - Solo5 boot info

As these structures do not affect isolation between subjects or subjects and the kernel, they are not generated by τ_0 but only included as binary data via XML command stream and build artifacts.

The structure generator tools are explained in section 5.2.9.

5.1.7 Image Creation

The system image composer assembles the final system image. This task is performed by τ_0 static introduced in the previous section 4.1. The usage of it is specified in 5.2.8.

5.2 Core Tools

This section describes the tools which form the core of the Muen toolchain.

²<https://github.com/Solo5/solo5>

5.2.1 Policy Merger

The merger tool mucfgmerge combines user-provided system policy files into a single XML document.

Name

mucfgmerge

Input

System configuration as XML, colon-separated list of include paths

Output

System policy in format source (merged)

This tool reads the system configuration and merges the specified system policy, hardware and platform files into a single file. To ease the creation of many similar variants of a system the provided system policy may use templates, expressions, conditionals and amend statements. The main processing-steps are:

1. Merge XIncludes of system policy into the system policy, i.e., insert the referenced files at the given location;
2. Merge hardware, additional hardware and platform specifications into the system policy. This includes merging the platform configuration section into the global configuration section;
3. Instantiate the templates, using the provided values and variable names;
4. Evaluate expressions, resulting in new configuration variables;
5. Replace all references to configuration variables with their value;
6. Evaluate conditionals, i.e., decide which sub-trees of the XML-tree to discard;
7. Evaluate amend statements, i.e., move sub-trees within the XML-tree.

The result is in policy source format and re-formatted so changes to the policy by subsequent build steps can be manually reviewed or visualized by diffing the files. In particular, the result does not contain any templates, expressions, conditionals or amend statements. The tool has debug modes that increase the verbosity of the output, in particular in case of errors.

5.2.2 Component Specification Joiner

The Muen component specification joiner adds component XML specifications to the component section of a specified system policy and writes the result to a designated output file. Each given component/library specification is loaded and validated against the component specification XML schema. If it is correct the content is added to the components section of the system policy specified as input file. If the given system policy does not yet contain a components section, it is created. The result is written to the file specified by the -o parameter. In-place processing is supported by passing in the same value for input and output file.

Name

mucfgcjoin

Input

System policy in format source, comma-separated list of component specs

Output

System policy in format source (joined)

5.2.3 Expander

The expander completes the user-provided system policy by creating or deriving additional configuration elements.

Name

mucfgexpand

Input

System policy in format source

Output

System policy in format A (expanded)

The Expander performs the following actions:

- Pre-check the system policy to make sure it is sound
- Expand channels
- Expand device resources
- Expand device isolation domains
- Expand kernel sections
- Expand minimal τ_0 subject
- Expand additional memory regions
- Expand hardware-/platform-related information
- Expand additional subject information
- Expand profile-specific information
- Expand scheduling information
- Post-check resulting policy

5.2.4 Allocator

The Allocator is responsible to assign a physical address to all global memory regions.

Name

mucfgalloc

Input

System policy in format A

Output

System policy in format B (allocated)

First, the Allocator initializes the physical memory view of the system based on the physical memory blocks specified in the XML hardware section. It then reserves memory that is occupied by pre-allocated memory elements (i.e. memory regions with a physical address or device memory). Finally it places all remaining memory regions in physical memory. In order to reduce the size of the final system image file-backed memory regions are placed at the start of memory.

5.2.5 Validator

The Validator performs additional checks that go beyond the basic restrictions imposed by the XML schema validation. For example it checks that the hardware provides an IOMMU device and that all references to subjects are resolvable. See [6](#) for a complete list of all executed checks. The tool aborts with a non-zero exit status and an explanatory message to the user if checks fail.

Name

mucfgvalidate

Input

System policy in format B

Output

None, raises exception on error

5.2.6 Hasher

The `mucfgmemhashes` tool is used to add memory integrity hashes to a given policy.

Name

`mucfgmemhashes`

Input

System policy in format B, colon-separated list of input directories containing build artifacts

Output

System policy in format B with memory integrity hashes

The tool appends a hash to all memory regions with fill and file content. It must run after all files have been generated by the structure generator tools.

The actual hash is generated using the SHA-256 algorithm and is intended to be used to verify the integrity of memory regions during runtime.

Note that no hashes are generated for sinfo memory regions. Since the hash information is exported via sinfo, and the sinfo region is itself part of the memory information of a subject, this hash would be self-referential.

The tool also replaces all occurrences of `hashRef` elements. A hash reference element instructs the tool to copy the hash element of the referenced memory region after message digest generation.

From an abstract point of view, the `hashRef` element is a way to link multiple memory regions by declaring that the hash of the content is the same. The hash may serve as an indicator on how to reconstruct the (initial) content of a memory region. This mechanism is used by e.g. the subject loader (SL) during subject init and reset operation. The subject loader expansion step remaps writable memory regions of the loadee (the subject under loader control) to SL and replaces the original regions with new ones containing a hash reference to the associated physical memory region. This way SL is able to determine the intended content of the target memory region by looking up the region in its sinfo page using the hash value as key.

5.2.7 Tau0 Command Stream Generator

The `mugentau0cmds` tool creates an XML command stream for τ_0 to let it compose the system image specified by the system policy given as input.

The tool reads the policy in format B and translates it to a sequence of commands as shown in the following listing:

```
1 <tau0>
2   <commands>
3     <addMemoryBlock address="16#0000#" size="157"/>
4     <addMemoryBlock address="16#0010_0000#" size="130816"/>
5     <addMemoryBlock address="16#2020_0000#" size="130564"/>
6     <addMemoryBlock address="16#4000_5000#" size="453932"/>
7     <addMemoryBlock address="16#bae9_f000#" size="256"/>
8     <addMemoryBlock address="16#baf9_f000#" size="96"/>
9     <addMemoryBlock address="16#0001_0000_0000#" size="3401216"/>
10    <addMemoryBlock address="16#ba3b_a000#" size="23"/>
11    <addMemoryBlock address="16#bb80_0000#" size="16896"/>
12    <addProcessor id="0" apicId="0"/>
13    <addProcessor id="1" apicId="2"/>
14    <addIoapic sid="16#f0f8#/"/>
15    <createLegacyDevice device="0"/>
16    <addIOPortRangeDevice device="0" from="16#03c0#" to="16#03df#/"/>
17    <addMemoryDevice device="0" caching="WC" address="16#000a_0000#" size="32"/>
18    <activateDevice device="0"/>
19    <createLegacyDevice device="1"/>
20    <addIOPortRangeDevice device="1" from="16#0060#" to="16#0060#/"/>
21    <addIOPortRangeDevice device="1" from="16#0064#" to="16#0064#/"/>
22    <addIRQDevice device="1" irq="1"/>
23    <addIRQDevice device="1" irq="12"/>
24    <activateDevice device="1"/>
25    <createLegacyDevice device="2"/>
26    <addIOPortRangeDevice device="2" from="16#0070#" to="16#0071#/"/>
27    <activateDevice device="2"/>
28    <createLegacyDevice device="3"/>
29    <addIOPortRangeDevice device="3" from="16#0cf9#" to="16#0cf9#/"/>
30    <addIOPortRangeDevice device="3" from="16#0404#" to="16#0404#/"/>
31    <activateDevice device="3"/>
32    <createLegacyDevice device="4"/>
33    <addMemoryDevice device="4" caching="UC" address="16#fec0_0000#" size="1"/>
34    <activateDevice device="4"/>
```

Listing 5.1: τ_0 Command Stream

As τ_0 strictly enforces certain invariants, the system must be constructed in a way not to violate these invariants. For example, before memory can be typed as being a VT-d root table, this memory must be cleared. Otherwise the memory typing model of τ_0 is violated.

The mugentau0cmds tool must take this into consideration when iterating over the resources specified in the input system policy and generating commands which instruct τ_0 to create the specified system.

Name

mugentau0cmds

Input

System policy in format B

Output

XML command stream for τ_0 static

5.2.8 Tau0 Static

The τ_0 static component serves as an image composer during integration. The concept and motivation of this approach is described in chapter 4.

Name

tau0_main

Input

XML command stream, colon-separated list of input directories containing build artifacts

Output

Muen system image

Output format

Command Stream Loader (CSL) image³, bootable by any compliant bootloader.

If a command is received which violates a constraint enforced by τ_0 static, the tool aborts system image construction, displays an error message and returns with a non-zero exit status.

5.2.9 Structure Generators

These tools do not change the policy and use it read-only.

MSR Stores Generator

Generate MSR store for each subject with MSR access.

Name

mugenmsrstore

Input

System policy in format B

Output

MSR store files of subjects in binary format

Output format

Intel SDM Vol. 3C, "24.8.2 VM-Entry Controls for MSRs" and Intel SDM Vol. 3C, "24.7.2 VM-Exit Controls for MSRs".

The tool generates MSR stores for each subject. The MSR store is used to save/load MSR values of registers not implicitly handled by hardware on subject exit/resumption.

MSR stores are used by hardware (VT-x) to enforce isolation of MSR (i.e. subjects that have access to the same MSRs cannot transfer data via these registers).

³<https://www.code-labs.ch/download/bsbsc-spec.pdf>

ACPI Tables

Generate ACPI tables for all Linux subjects.

Name

mugenacpi

Input

System policy in format B

Output

ACPI tables of all Linux subjects

Output format

Advanced Configuration and Power Interface (ACPI) Specification⁴

ACPI tables are used to announce available hardware to VM subjects. A set of tables consists of an RSDP, XSDT, FADT and DSDT table. See the ACPI specification for more information about a specific table.

Linux Zero Pages

Generate Zero Pages for all Linux subjects.

Name

mugenzp

Input

System policy in format B

Output

Zero pages of all Linux subjects

Output format

Linux Boot Protocol⁵

Zero Page⁶

The so-called Zero Page (ZP) exports information required by the boot protocol of the Linux kernel on the x86 architecture. The kernel uses the provided information to retrieve settings about its runtime environment:

- Type of bootloader
- Map of physical memory (e820 map)
- Address and size of initial ramdisk(s)
- Kernel command line parameters

Solo5 Boot Info

Generate Solo5 boot info structures for MirageOS unikernels⁷ running on the Solo5 platform.

Name

mugensolo5

Input

System policy in format B

Output

Solo5 boot info for all MirageOS subjects

Output format

struct hvt_boot_info⁸

⁴<http://www.acpi.info/DOWNLOADS/ACPIspec50.pdf>

⁵<https://www.kernel.org/doc/Documentation/x86/boot.txt>

⁶<https://www.kernel.org/doc/Documentation/x86/zero-page.txt>

⁷<https://mirage.io>

⁸https://github.com/Solo5/solo5/blob/master/include/solo5/hvt_abi.h

The boot info structure exports information required by Solo5. The unikernel uses the provided information to retrieve settings about its runtime environment:

- Memory size in bytes
- Address of end of unikernel
- CPU cycle counter frequency, Hz
- Address of command line (C string)
- Address of application manifest

Kernel Source Specifications

Generate source specifications used by kernel.

Name

`mugenspec`

Input

System policy in format B

Output

Source specifications in SPARK, C and GPR format

Gathers data from the system policy to generate various source files in SPARK, C and GNAT project file (GPR) format. Created output includes constant values for memory addresses, device resources, scheduling plans, etc. See the description of the `Skp` package hierarchy in the Muen Kernel Specification document [2] for the exact information these packages provide.

Component Source Specifications

Process component description and generate source specifications from it. Write processed description to specified output file.

Name

`mucgenspec`

Input

Component description in XML, colon-separated list of include paths

Output

Component source specifications in SPARK, processed component description in XML

The component spec generation tool processes the given component description by evaluating XIncludes, boolean expressions and resolving conditional parts. Furthermore, it performs substitutions of attributes with configuration values.

It also generates Ada/SPARK packages containing constants of the declared logical component resources. The generated specifications can be used in the component source code to access the declared resources.

The resulting processed component description is written to the given output location.

Subject Info (sinfo)

Generate subject information data for each subject.

Name

`mugensinfo`

Input

System policy in format B

Output

Subject info data in binary format

Output format

As specified in [1] and `common/musinfo/musinfo.ads`

The Sinfo page is used to export subject information data extracted from the system policy to subjects. Currently, information about available memory regions, communication channels, events, vectors and assigned PCI devices is provided.

5.3 Additional Tools

This section lists additional helper tools which simplify the process of generating and validating a Muen system.

5.3.1 Kernel ELF Checker

The `muchekelf` tool enforces that the format of a given Muen kernel ELF binary matches the kernel memory layout specified in a system policy. Furthermore, the ELF kernel entry point is compared to the expected value.

Size, VMA (Virtual Memory Address) and permissions of binary ELF sections are validated against kernel memory regions defined in the policy. The following table lists the correspondence of ELF section names to logical kernel memory region names.

ELF Section	Memory Name
<code>.text</code>	<code>kernel_text</code>
<code>.data</code>	<code>kernel_data</code>
<code>.rodata</code>	<code>kernel_ro</code>
<code>.bss</code>	<code>kernel_bss</code>
<code>.globaldata</code>	<code>kernel_globaldata</code>

5.3.2 Stack Usage Checker

The `muchekstack` tool statically calculates the worst-case stack usage of a native Ada/SPARK component or the Muen kernel compiled with the `-fcallgraph-info` switch⁹.

The tool takes a GNAT project file and a stack limit in bytes as input. All control-flow information (.ci) files found in the object directory of the main project and all of its dependencies are parsed. Once the control-flow graph is constructed the maximum stack usage of each subprogram is calculated and checked against the user-specified limit. The tool exits with a failure if a stack usage exceeding the limit is detected.

Note that the tool is not applicable to arbitrary software projects as it does not handle dynamic/unbounded stack usage and recursion. In the context of the Muen project these cases can not occur since they are prohibited by the following restriction pragmas:

- `No_Recursion`
- `No_Secondary_Stack`
- `No_Implicit_Dynamic_Code`

Additionally, the `-Wstack-usage` compiler switch warns about potential unbounded stack usage.

5.3.3 Hardware Config Generator

The `mugenhwcfg`¹⁰ tool has been created to automate the process of gathering all necessary hardware information. To collect data for a new target hardware all that is required is to run the tool on a common Linux distribution¹¹. See the project README for more information.

Name

`mugenhwcfg`

Input

`None`

Output

Hardware description in `output.xml`

⁹https://www.adacore.com/uploads/technical-papers/Stack_Analysis.pdf

¹⁰<https://git.codelabs.ch/?p=muen/mugenhwcfg.git>

¹¹<https://github.com/roburio/mugenhwcfg-live>

The tool is implemented in a way to extract as much information from the system and generate a hardware configuration even if problems are encountered. The aim is to assist the integrator as much as possible in writing a hardware configuration for the target hardware.

Therefore, the tool only fails with a non-zero exit status and no output if essential required data can not be extracted from the system. Other problems are reported in the potentially incomplete `output.xml` file as XML comments, making the encountered problems on the actual machine evident. The following snippet provides an example of such a warning comment in the header of the generated `output.xml` file:

```
* WARNING *: Unable to resolve device class 0c80. Please update pci.ids  
(-u) and try again
```

The comments should be rather self-explanatory. In this case, the problem is only a minor issue since the tool was simply unable to resolve a device class number to a human-readable string.

The next example has more consequences:

```
* WARNING *: Skipping invalid IRQ resource for device 0000:00:1f.3: None
```

This has the effect that no IRQ resource is appended in the specification of the device exhibiting this problem. While the device can still be assigned to a subject, it is missing the IRQ element and as a result the IRQ resource itself. It can be assumed that this leads to problems with the driver interacting with the device. For proper operation, it is the policy writer's task to rectify the hardware specification by determining the correct configuration manually.

5.3.4 Component Binary Splitter

The `mucbinsplit` tool splits component binaries into multiple files, one per ELF section.

Name

`mucbinsplit`

Input

Component description in XML, Component ELF binary

Output

Binary files corresponding to ELF sections, processed component description in XML

The component binary splitter tool processes component binaries and creates a separate file for each ELF section. The component XML description is extended by adding a file-backed memory region for each ELF section with the appropriate virtual mapping address, size and access rights. The RIP value is set to the ELF entry point of the component binary.

The resulting processed component description is written to the given output location while the binary section files are written to the specified output path.

Chapter 6

Policy Validation

Prior to operate on the policy, any tool outlined in the toolchain section 5.2 checks all required preconditions by running *validator* procedures. For example a tool accessing physical devices via subject logical device references will execute a validator checking such references for validity.

Before the policy is used to generate system structures like sinfo regions, or the command stream for τ_0 , the expanded policy in format B is validated by executing a comprehensive set of checks. This is done by the `mucfgvalidate` tool outlined in section 5.2.5.

The following sections list the various checks executed by `mucfgvalidate` and the other Muen build tools in the toolchain.

6.1 Configuration

The following checks are performed to guarantee correctness of configuration options in the system policy.

- Validate config variable name uniqueness.
- Check that all booleans defined in config contain a valid value.
- Check that all integers defined in config contain a valid value.
- Check that all expression config variable references are valid.
- Check that all integers defined in expressions contain a valid value.
- Check that all booleans defined in expressions contain a valid value.
- Check that all conditional config variable references are valid.

6.2 Devices

The following checks are performed to guarantee that hardware devices are correctly configured in the system policy.

- Validate that devices referenced by logical devices exist.
- Validate that device names (including device aliases/classes) are unique.
- Validate that all physical IRQs are unique.
- Validate that physical device IRQs referenced by logical IRQs exist.
- Validate that ISA IRQs fulfill their constraints.
- Validate that PCI LSI IRQs fulfill their constraints.
- Validate that PCI MSI IRQs fulfill their constraints.
- Validate that PCI MSI IRQs are consecutive.

- Validate that physical IRQ names are unique per device.
- Validate that all I/O start ports are smaller than end ports.
- Validate that physical I/O ports referenced by logical I/O ports exist.
- Validate that all physical I/O ports are unique.
- Validate that physical I/O port names are unique per device.
- Validate that device memory names are unique per device.
- Validate that device memory referenced by logical device memory exists.
- Validate that PCI device bus, device, function triplets are unique.
- Validate that logical device references of each subject do not refer to the same physical device.
- Validate that PCI device reference bus, device, function triplets are unique per subject.
- Validate that all device references specifying a bus, device, function triplet are references to physical PCI devices.
- Validate that all device references to PCI multi-function devices belong to the same subject and have the same logical device number.
- Validate that all device references not specifying a bus, device, function triplet are references to physical legacy (non-PCI) devices.
- Validate that all logical PCI devices specify bus number zero.
- Validate that all IOMMU memory-mapped IO regions have a size of 4K.

6.3 Device Domains

The following checks are performed to guarantee that IOMMU device domains are correctly configured in the system policy.

- Validate that domain device references are unique.
- Validate that no virtual memory regions of a domain overlap.
- Validate that domain memory referenced by subjects is mapped at the same virtual address.
- Validate memory type of physical memory referenced by domains.
- Validate that each device referenced by a device domain is a PCI device.
- Validate that each device domain has a physical PT memory region.
- Validate that each PCI bus has a physical VT-d context memory region.

6.4 Events

The following checks are performed to guarantee that events are correctly configured in the system policy.

- Check that all physical event names are unique.
- Check that each global event has associated sources and one target.
- Check subject event references.
- Validate that there are no self-references in subject's event notification entries.
- Validate that notification entries switch to a subject running on the same core and in the same scheduling group.

- Validate that target subjects of IPI notification entries run on different logical CPUs.
- Validate that target event IDs as well as logical names are unique.
- Validate that source event IDs as well as logical names are unique per group.
- Check source event ID validity.
- Check that source event IDs of the VMX Exit group are all given or a default is specified.
- Check that self events provide a target action.
- Check that kernel-mode events have an action specified.
- Check that system-related actions are only used with kernel-mode events.
- Check that level-triggered IRQs have a corresponding unmask IRQ event.

6.5 Files

The following file-specific checks are performed.

- Check existence of files referenced in XML policy.
- Check if files fit into corresponding memory region.

6.6 Hardware

The following checks are performed on the hardware section of the policy.

- Validate that memory regions fit into available hardware memory.
- Validate that no memory blocks overlap.
- Validate that the size of memory blocks is a multiple of page size.
- Validate that PCI config space address and size are specified if PCI devices are present.
- Validate that the hardware provides enough physical CPU cores.
- Validate that the processor CPU sub-elements are correct.
- Validate that at least one I/O APIC device is present.
- Validate that all I/O APICs have a valid source ID capability.
- Validate that at least one and at most eight IOMMU devices are present.
- Validate that all IOMMUs have the AGAW capability set correctly and that multiple IOMMUs specify the same value.
- Validate that all IOMMUs have correct register offset capabilities.
- Check that the hardware contains a system board device providing the expected configuration.

6.7 Kernel

The following kernel-specific checks are performed on the policy.

- Validate that all CPU-local data section virtual addresses are equal.
- Validate that all CPU-local BSS section virtual addresses are equal.
- Validate that all global data section virtual addresses are equal and that the expected number of mappings exists.
- Validate that all stack virtual addresses are equal.
- Validate that all crash audit mappings exist and that their virtual addresses are equal.
- Validate that every kernel has a stack and interrupt stack region mapped and both regions are guarded by unmapped pages below and above.
- Validate that all IOMMU memory-mapped IO regions are consecutive.
- Validate that each active CPU has a memory section.
- Validate that no virtual memory regions of the kernel overlap.
- Validate that the system board is referenced in the kernel logical devices section and that it provides a logical reset port.
- Validate that the debug console device and its resources matches the kernel diagnostics device specified in the platform section.

6.8 Memory

The following checks are performed to verify that the memory is correctly configured in the system policy.

- Validate that a VMXON region exists for every specified kernel.
- Validate size of VMXON regions.
- Validate that VMXON regions are in low-mem.
- Validate that all VMXON regions are consecutive.
- Validate that a VMCS region exists for each declared subject.
- Validate size of VMCS regions.
- Validate that physical memory region names are unique.
- Validate that physical memory referenced by logical memory exists.
- Validate that all physical memory addresses are page aligned.
- Validate that all virtual memory addresses are page aligned.
- Validate that all memory region sizes are multiples of page size.
- Validate kernel or subject entities encoded in physical memory names (e.g. linux|zp or kernel_0|vmxon).
- Validate that no physical memory regions overlap.
- Validate that an uncached crash audit region is present.
- Validate that a kernel data region exists for every CPU.
- Validate that a kernel BSS region exists for every CPU.

- Validate that a kernel stack region exists for every CPU.
- Validate that a kernel interrupt stack region exists for every CPU.
- Validate that a kernel PT region exists for every CPU.
- Validate that kernel PT regions are in the first 4G.
- Validate that scheduling info regions are mapped by the kernel running subjects of that scheduling partition. Also verify that the kernel mapping is at the expected virtual location.
- Validate that a subject state memory region with the expected size exists for every subject.
- Validate that a subject interrupts memory region with the expected size exists for every subject.
- Validate that memory of type kernel is only mapped by kernel or Tau0.
- Validate that memory of type system is not mapped by any entity.
- Validate that memory of type 'device' (e.g. device_rmrr) is only mapped by device domains.
- Validate that subject state memory regions are mapped by the kernel running that subject. Also verify that the kernel mapping is at the expected virtual location.
- Validate that subject interrupts memory regions are mapped by the kernel running that subject. Also verify that the kernel mapping is at the expected virtual location.
- Validate that subject MSR store memory regions are mapped by the kernel running that subject. Also verify that the kernel mapping is at the expected virtual location.
- Validate that subject timed event memory regions are mapped by the kernel running that subject. Also verify that the kernel mapping is at the expected virtual location.
- Validate that subject VMCS regions are mapped by the kernel running that subject. Also verify that the kernel mapping is at the expected virtual location.
- Validate that subject FPU state regions are mapped by the kernel running that subject. Also verify that the kernel mapping is at the expected virtual location.
- Validate that a subject FPU state memory region with the expected size exists for every subject.
- Validate that a subject timed event memory region with the expected size exists for every subject.
- Validate that a subject I/O Bitmap region with the expected size exists for every subject.
- Validate that a subject MSR Bitmap region with the expected size exists for every subject.
- Validate that a subject MSR store memory region exists for each subject that accesses MSR registers not managed by VMCS.
- Validate that a subject pagetable memory region exists for each subject.
- Validate that a scheduling info memory region exists for each scheduling partition.
- Validate that subjects map the scheduling info region of their associated scheduling partition.
- Validate that subject state, timed event and pending interrupts memory regions are only mapped writable by subjects running on the same CPU core or by siblings.
- Validate size of VT-d root table region.
- Validate size of VT-d context table region.
- Validate that a VT-d root table region exists if domains are present.
- Validate that a VT-d interrupt remapping table region exists.

6.9 Model Specific Registers (MSR)

The following checks are performed to verify Model Specific Register (MSR) specifications in the system policy.

- Validate that all MSR start addresses are smaller than end addresses.
- Validate that subject MSRs are in the allowed list:
 - IA32_SYSENTER_CS/ESP/EIP
 - IA32_DEBUGCTL
 - IA32_EFER/STAR/LSTAR/CSTAR/FMASK
 - IA32_FS_BASE/GS_BASE/KERNEL_GS_BASE
 - MSR_PLATFORM_INFO
 - IA32_THERM_STATUS
 - IA32_TEMPERATURE_TARGET
 - IA32_PACKAGE_THERM_STATUS
 - MSR_RAPL_POWER_UNIT
 - MSR_PKG_POWER_LIMIT
 - MSR_PKG_ENERGY_STATUS
 - MSR_DRAM_ENERGY_STATUS
 - MSR_PP1_ENERGY_STATUS
 - MSR_CONFIG_TDP_CONTROL
 - IA32_PM_ENABLE
 - IA32_HWP_CAPABILITIES
 - IA32_HWP_REQUEST

6.10 Platform

The following checks are performed to verify the correctness of the platform configuration in the system policy.

- Validate that physical devices referenced by device aliases exist.
- Validate that physical device resources referenced by device aliases exist.
- Validate that physical devices referenced by device classes exist.
- Validate that subject devices that reference an alias only contain resources provided by the device alias.
- Validate that the physical device and resources referenced by the kernel diagnostics device exists.
- Validate that the kernel diagnostics device resources match the requirements of the specified diagnostics type.

6.11 Scheduling

The following checks are performed to verify the correctness of the scheduling configuration in the system policy.

- Validate that scheduling partition IDs are unique.
- Validate that scheduling group IDs are unique.
- Validate that each major frame specifies the same number of CPUs.

- Validate that scheduling partitions are scheduled in at least one minor frame and that all minor frame references are on the same logical CPU.
- Validate subject references.
- Validate that subjects are scheduled on the correct logical CPU.
- Validate that subjects are part of exactly one scheduling group.
- Validate that all subjects of a scheduling group are runnable.
- Validate tick counts in major frame.
- Validate that barrier IDs do not exceed barrier count and are unique.
- Validate that barrier sizes do not exceed the number of logical CPUs.
- Validate that the barrier sizes and count of a major frame corresponds to the minor frame synchronization points.
- Validate that minor frame barrier references are valid.
- Validate partition references in minor frames.

6.12 Subjects

The following checks are performed to verify the correctness of the subject configuration in the system policy.

- Validate subject name uniqueness.
- Validate subject CPU ID.
- Validate uniqueness of global subject IDs.
- Validate per-CPU uniqueness of local subject IDs.
- Validate memory types of memory mappings (ie. allow access by subjects).
- Validate that no subject references an IOMMU device.
- Validate that all subjects are runnable, i.e. referenced in a scheduling group.
- Validate that subject scheduling group IDs match values as determined by the scheduling plan and handover events.
- Validate that logical names of subject memory regions are unique.
- Validate that logical names of subject devices are unique.
- Validate that IRQ vector numbers of PCI device references with MSI enabled are consecutive.
- Validate that logical names of subject unmask IRQ events conform to the naming scheme (`unmask_irq_${IRQNR}`) and that the unmask number matches the physical IRQ.
- Validate that no virtual memory regions of a subject overlap.
- Validate that multiple initramfs regions are consecutive.
- Validate that no subject has write access to the crash audit region.
- Validate that subject device mmconf mappings are correct.
- Validate that the VMX controls conform to the checks specified in Intel SDM Vol. 3C, "26.2.1 Checks on VMX Controls".
- Validate that the Pin-Based VM-Execution controls meet the requirements for the execution of Muen.

- Validate that the Processor-Based VM-Execution Controls meet the requirements for the execution of Muen.
- Validate that the secondary Processor-Based VM-Execution Controls meet the requirements for the execution of Muen.
- Validate that the VM-Exit Controls meet the requirements for the execution of Muen.
- Validate that the VM-Entry Controls meet the requirements for the execution of Muen.
- Validate that the VMX CR0 guest/host masks meet the requirements for the execution of Muen.
- Validate that the VMX CR4 guest/host masks meet the requirements for the execution of Muen.
- Validate that the VMX Exception bitmap meet the requirements for the execution of Muen.

Chapter 7

Policy Structure

7.1 Policy Schema Documentation

7.1.1 systemType

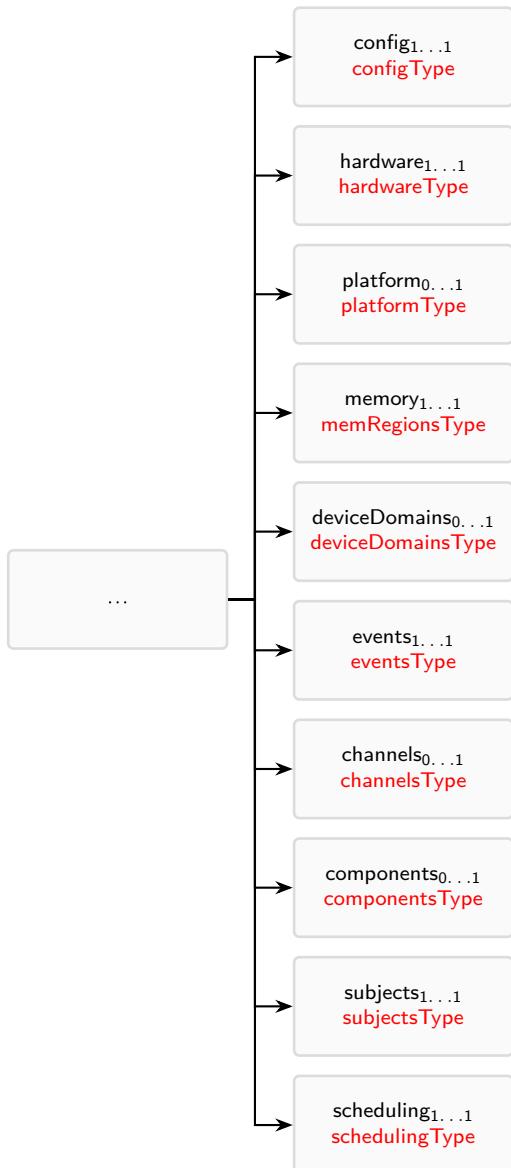
A Muen system policy specifies all hardware resources such as physical memory, devices, CPU time, etc and how these resources are accessed by the separation kernel, the subjects and devices.

The `system` section is the top-level element in the Muen system policy. It contains various sub-elements which specify all aspects of a concrete system.

This is the *source format* of the Muen system policy. It allows for abstractions, such as channels, which are broken down into their constituent parts by the toolchain in format A and B accordingly.

See line 3 and following in listing 8.1 on page 97 for an annotated system policy example.

Structure



7.1.2 configType

The purpose of a config section is to specify configuration values which parameterize a system or a component. It allows to declare boolean, string and integer values. The following sections in the system policy provide support for configuration values:

- System
- Platform
- Component

During the build process, configuration values provided by the platform are merged into the global system configuration. Component configuration values allow the parameterization of component-local functionality.

Besides component parameterization, configuration options can be used in `if` conditionals, as shown in the following example.

```
1  <if variable="xhcidbg_enabled" value="true">
2    ...
3  </if>
```

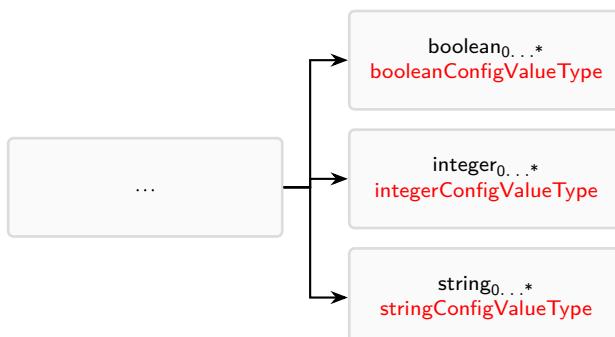
If the type of the referenced variable is 'string' the comparison is case-sensitive. A second use case is XML attribute value expansion as follows:

```
1 <channel name="debuglog" size="$logchannel_size"/>
```

The `size` attribute value is not specified directly, but parameterized via an integer configuration option.

See line 17 in listing 8.1 for an example config section.

Structure



7.1.3 booleanConfigValueType

Configuration option for values in boolean format.

Attributes

Name	Type	Use
name	nameType	optional Name of the configuration option.
value	booleanType	optional Value of the configuration option.

7.1.4 nameType

Base: xs:string

The `nameType` is used to give (unique) names to elements.

Restrictions

minimal length = 1, maximal length = 63

7.1.5 booleanType

Base: xs:string

Boolean type.

Restrictions

values:

- true
- false

7.1.6 integerConfigValueType

Configuration option for values in integer format.

Attributes

Name	Type	Use
name	nameType	optional Name of the configuration option.
value	xs:integer	optional Value of the configuration option.

7.1.7 stringConfigValueType

Configuration option for values in string format.

Attributes

Name	Type	Use
name	nameType	optional Name of the configuration option.
value	xs:string	optional Value of the configuration option.

7.1.8 hardwareType

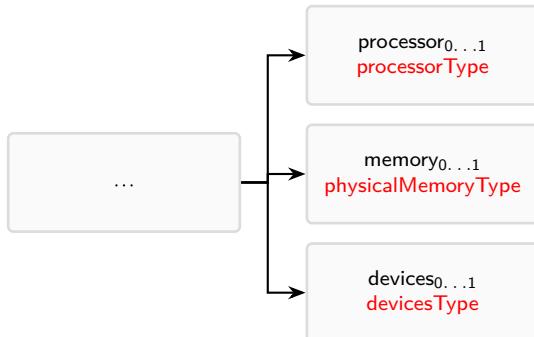
Systems running the Muen SK perform static resource allocation at integration time. This means that all available hardware resources of a target machine must be defined in the system policy in order for these resources to be allocated to subjects.

The hardware element is the top-level element of the hardware specification in the system policy. Information provided by a hardware description includes the amount of available physical memory blocks including reserved memory regions (RMRR), the number of physical CPU cores and hardware device resources.

The Muen toolchain provides a handy tool to automate the cumbersome process of gathering hardware resource data from a running Linux system: mugenhwcfg¹.

See line 81 in listing 8.1 for an example hardware section.

Structure



7.1.9 processorType

The processor element specifies the number of CPU cores, the processor speed in kHz and the Intel VMX preemption timer rate.

Since Intel CPUs can have arbitrary APIC identifiers, the APIC IDs of all physical CPUs are enumerated here. The APIC ID is required for interrupt and IPI routing.

¹<https://git.codelabs.ch/?p=muen/mugenhwcfg.git>

The processor element also lists register values for all CPUID leaves of the hardware target, and some MSR values of interest.

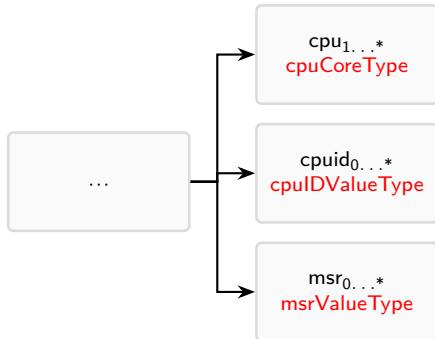
See line 100 in listing 8.1 for an example processor element. The cpu elements must fulfill the following constraints to be valid:

- A node exists for every physical core of the system
- The optional cpuid attribute of all elements must be consecutive
- If specified, a node with cpuid value 0 must exist
- A node with apicId value 0 must exist and, if specified, it must have a cpuid value within the active CPU range, i.e. the BSP is part of the system scheduling plan
- All apicId attributes must have even numbers

Attributes

Name	Type	Use
cpuCores	xs:positiveInteger	required Number of available CPU cores. Note that this value designates physical, hardware cores, not Hyper-Threading (HT) threads. HT is disabled on Muen.
speed	xs:positiveInteger	required Tick rate of CPU cores in kHz.
vmxTimerRate	vmxTimerRateType	required The VMX-preemption timer counts down at a rate proportional to that of the timestamp counter (TSC). This value specifies this proportion, see Intel SDM Vol. 3C, "25.5.1 VMX-Preemption Timer" for more details.

Structure



7.1.10 cpuCoreType

Specification of one physical CPU core.

Attributes

Name	Type	Use
apicId	xs:unsignedByte	required CPU local APIC ID, see Intel SDM Vol. 3A, "10.4.6 Local APIC ID".
cpuid	xs:unsignedByte	optional Unique CPU ID.

7.1.11 cpuidValueType

Register values for a CPUID leaf, see Intel SDM Vol. 2A, "3.2 Instructions (A-L)", CPUID.

Attributes

Name	Type	Use
leaf	word32Type	required CPUID leaf.
subleaf	byteType	required CPUID subleaf.
eax	word32Type	required EAX register value of this leaf.
ebx	word32Type	required EBX register value of this leaf.
ecx	word32Type	required ECX register value of this leaf.
edx	word32Type	required EDX register value of this leaf.

7.1.12 word32Type

Base: word64Type < xs:string
32-bit machine word.

Restrictions

value ≤ 13

7.1.13 byteType

Base: xs:string
Machine octet (8-bits).

Restrictions

Pattern = 16#[0-9a-fA-F]2#

7.1.14 msrValueType

Register value of an MSR of interest.

Attributes

Name	Type	Use
address	word32Type	required MSR address.
name	xs:string	required Name of MSR.
regval	word64Type	required Register value.

7.1.15 word64Type

Base: xs:string
64-bit machine word.

Restrictions

Pattern = 16#[0-9a-fA-F]4(_([0-9a-fA-F]4))0,3#

7.1.16 vmxTimerRateType

Base: xs:nonNegativeInteger
VMX-preemption timer count down rate.

Restrictions

value ≤ 31

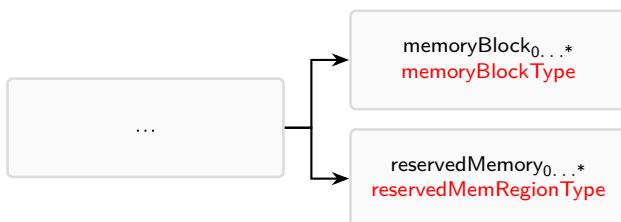
7.1.17 physicalMemoryType

The hardware memory element specifies the available physical memory blocks including reserved memory regions (RMRR, see Intel VT-d Specification, "8.4 Reserved Memory Region Reporting Structure").

Only memory blocks reported by the BIOS E820 map as non-reserved must be configured in this section, e.g. *usable* or *ACPI NVS*, *ACPI data*.

See line 165 in listing 8.1 for an example memory element.

Structure



7.1.18 memoryBlockType

Base: memoryBlockBaseType

Consecutive block of memory provided by the hardware.

Attributes

Name	Type	Use
name	nameType	required Name of memory block.
physicalAddress	word64Type	required Start address of memory block.
size	memorySizeType	required Size of memory block.
allocatable	booleanType	optional Indication to a physical memory allocator that this block allows allocation of physical memory regions. If this attribute is false, an allocator should only place fixed memory regions in this range, i.e. memory regions with the physicalAddress attribute set by the integrator. Note that host physical memory below 1 MiB is considered special, the attribute must be set to false. Only unmapped memory of type system is allowed in that special memory block.

7.1.19 memorySizeType

Base: word64Type < xs:string

The memorySizeType is used to declare memory sizes.

Restrictions

no restriction

7.1.20 reservedMemRegionType

Base: memoryBlockBaseType

A reservedMemory element is a special memory block declaration. It specifies a reserved memory region as outlined in the Intel VT-d Specification, "8.4 Reserved Memory Region Reporting Structure" (RMRR).

Reserved memory regions are BIOS allocated memory ranges that may be DMA targets for certain legacy device use-cases. Devices that require access to such a region refer to it by name.

See line 181 in listing 8.1 for an example RMRR element.

Attributes

Name	Type	Use
name	nameType	required
	Name of memory block.	
physicalAddress	word64Type	required
	Start address of memory block.	
size	memorySizeType	required
	Size of memory block.	

7.1.21 devicesType

The devices element enumerates all devices provided by the hardware platform. Different kinds of devices, be it PCI(e) or legacy (non-PCI), can be declared in this section.

See line 194 in listing 8.1 for an example devices enumeration.

Attributes

Name	Type	Use
pciConfigAddress	word64Type	optional
	Physical base address of the PCI configuration space region.	
pciConfigSize	word64Type	optional
	Size of the PCI configuration space region.	

Structure



7.1.22 deviceType

Base: deviceBaseType

The device element specifies a physical device and its associated resources. There are three main device resource types:

- IRQ
- I/O port range
- Memory

The presence of a PCI element indicates whether the device is a PCI or a legacy device.

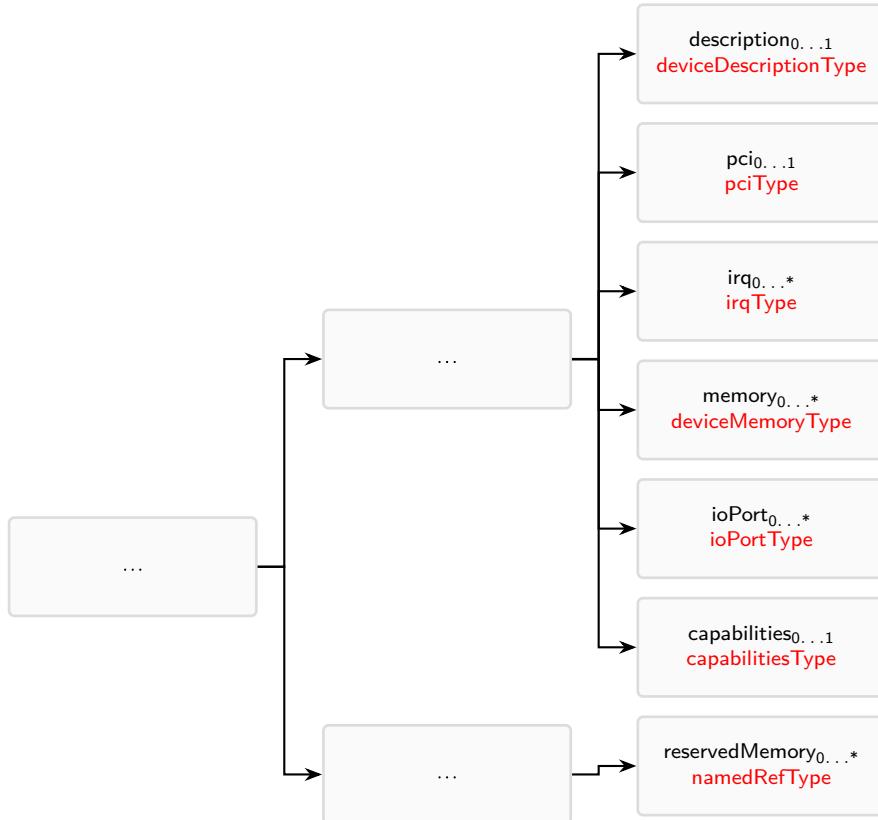
Capabilities can be used to convey additional device-specific information. The base address of the memory mapped PCI config space is defined by the pciConfigAddress attribute.

See line 200 in listing 8.1 for an example device declaration.

Attributes

Name	Type	Use
name	nameType	required
	Unique device name.	

Structure



7.1.23 namedRefType

The namedRefType is used to reference a named element in the policy.

Attributes

Name	Type	Use
ref	nameType	required Name of referenced element.

7.1.24 deviceDescriptionType

Base: xs:string

Device description (free text).

Restrictions

no restriction

7.1.25 pciType

Base: pciAddressType

PCI(e) devices are specified using the pci element.

The element provides the following information:

- PCI device address (BDF)
- Identification
- IOMMU group information

The location of the PCI device in the PCI topology is specified by the Bus, Device, Function triplet (BDF).

See line 334 in listing 8.1 for an example PCI element declaration.

Attributes

Name	Type	Use
bus	byteType	required PCI Bus number.
device	pciDeviceNumberType	required PCI Device number.
function	pciFunctionNumberType	required PCI Function number.

Structure



7.1.26 deviceIdentificationType

The identification element specifies the PCI device class, device, revision and vendor ID.

For more information, consult the PCI Local Bus Specification, "Configuration Space Decoding".

See line 349 in listing 8.1 for an example PCI identification.

Attributes

Name	Type	Use
classcode	word16Type	required PCI device class.
vendorId	word16Type	required PCI vendor ID.
deviceId	word16Type	required PCI device ID.
revisionId	byteType	required PCI device revision ID.

7.1.27 word16Type

Base: word64Type < xs:string

16-bit machine word.

Restrictions

length = 8

7.1.28 iommuGroupType

Devices in the same IOMMU group cannot be properly isolated from each other because they may perform inter-device transactions directly, without going through the IOMMU.

Note that this information is currently not used by the toolchain. It is a hint to the system integrator whether two devices can be properly isolated from each other or not.

See line 358 in listing 8.1 for an example IOMMU group declaration.

Attributes

Name	Type	Use
id	xs:nonNegativeInteger	required IOMMU group number.

7.1.29 pciDeviceNumberType

Base: xs:string

PCI Device number.

Restrictions

Pattern = 16#[0|1][0-9a-fA-F]#

7.1.30 pciFunctionNumberType

Base: xs:nonNegativeInteger

PCI Function number.

Restrictions

value \leq 7

7.1.31 irqType

The `irq` element specifies a device IRQ resource.

The specified IRQ number is one of:

- Legacy IRQ (ISA)
Range 0 .. 15.
- PCI INTx IRQ, line-signaled
Range 0 .. Max_LSI_IRQ, whereas Max_LSI_IRQ is defined by the hardware I/O APIC configuration `gsi_base + max_redirection_entry` of I/O APIC with `max(gsi_base)`. `gsi_base` and `max_redirection_entry` are I/O APIC device capabilities.

`msi` sub-elements are present if the device supports MSI interrupts. The element count designates the number of supported MSI interrupts.

See line 237 in listing 8.1 for an example device IRQ declaration.

Attributes

Name	Type	Use
name	nameType	required Name of device IRQ resource.
number	irqNumberType	required Legacy or PCI line-based IRQ.

Structure



7.1.32 msiIrqType

There are two different interrupt types which devices may trigger: legacy/PCI LSI IRQs and Message Signaled Interrupts (MSI). The legacy/PCI LSI IRQ is specified by the number attribute

of the `irq` element. For MSIs, each `msi` element defines an MSI IRQ that may be assigned to subjects. Each MSI may be individually routed.

See line 400 in listing 8.1 for example device MSI elements.

Attributes

Name	Type	Use
<code>name</code>	<code>nameType</code>	required Name of MSI resource.

7.1.33 irqNumberType

Base: `xs:nonNegativeInteger`

IRQ number. High IRQs are reserved for kernel usage.

Restrictions

`value ≤ 220`

7.1.34 deviceMemoryType

Base: `memoryBlockBaseType`

A device memory element specifies a memory region which is used to interact with the associated device.

For PCI devices, the specified region is programmed into one device BAR (Base Address Register) by system firmware. See the PCI Local Bus Specification or the PCI Express Base Specification for more details.

See line 218 in listing 8.1 for an example device memory declaration.

Attributes

Name	Type	Use
<code>name</code>	<code>nameType</code>	required Name of memory block.
<code>physicalAddress</code>	<code>word64Type</code>	required Start address of memory block.
<code>size</code>	<code>memorySizeType</code>	required Size of memory block.
<code>caching</code>	<code>cachingType</code>	required Device memory caching type.

7.1.35 cachingType

Base: `xs:string`

Memory caching type, see Intel SDM Vol. 3A, "11.3 Methods of Caching Available".

- Strong Uncacheable (UC)
- Write Combining (WC)
- Write Through (WT)
- Write Back (WB)
- Write Protected (WP)

Restrictions

`Pattern = UC|WC|WT|WB|WP`

7.1.36 ioPortType

The `ioPort` element specifies a device I/O port resource from `start` octet up to and including `end` octet. A single byte-accessed port is designated by specifying the same `start` and `end` values.

See line 228 in listing 8.1 for an example device IRQ declaration.

Attributes

Name	Type	Use
<code>name</code>	<code>nameType</code>	required Name of I/O port resource.
<code>start</code>	<code>word16Type</code>	required Start port of this resource.
<code>end</code>	<code>word16Type</code>	required End port of this resource.

7.1.37 capabilitiesType

List of device capabilities.

Structure



7.1.38 capabilityType

Base: `xs:string`

A device capability is used to assign additional information to a device. Such a capability might be used by the Muen toolchain to perform certain actions on devices with a given capability (e.g. `ioapic`). A system integrator may use this facility to define its own capabilities used by custom tools.

A capability element can have an optional value.

See line 290 in listing 8.1 for example capabilities.

Attributes

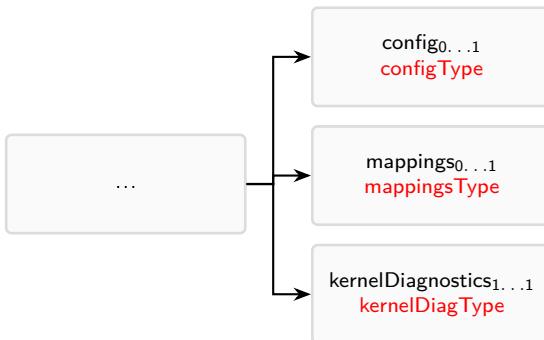
Name	Type	Use
<code>name</code>	<code>xs:string</code>	required Capability name (free text).

7.1.39 platformType

To enable an uniform view of the hardware resources across different physical machines from the system integrators perspective, the platform description layer is interposed between the hardware resource description and the rest of the system policy. This allows to build a Muen system for different physical target machines using the same system policy.

See line 556 in listing 8.1 for an example platform section.

Structure

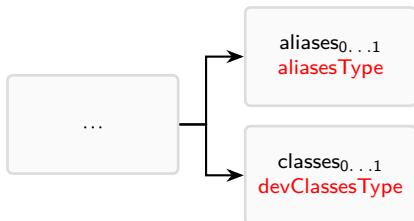


7.1.40 mappingsType

Platform device alias and class mappings section. Used to assign a stable name to a hardware device or to group (multiple) devices under a given name.

See line 565 in listing 8.1 for an example platform mappings section.

Structure



7.1.41 aliasesType

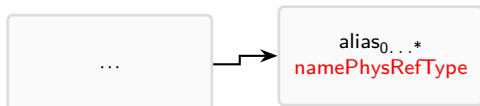
Aliases are a renaming mechanism for physical hardware devices and their resources. By using alias names in the system policy references to concrete hardware resources can be avoided. Additionally, aliases may be used to define a device which only contains a subset of the resources of the physical device. This can be achieved by only renaming the resources that the device alias should export.

See line 571 in listing 8.1 for an example aliases section.

Aliases are resolved in the following system policy sections.

- /system/subjects/subject/component/map
- /system/subjects/subject/devices/device
- /system/deviceDomains/domain/devices/device

Structure



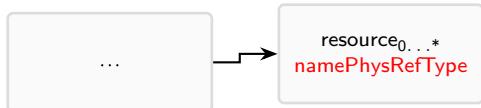
7.1.42 namePhysRefType

Named resource reference. Used for device aliases and device alias resource references.

Attributes

Name	Type	Use
name	nameType	required
physical	nameType	required Reference to physical device or device resource.

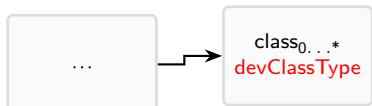
Structure



7.1.43 devClassesType

The `classes` element specifies a list of device classes.

Structure



7.1.44 devClassType

Device classes enable the grouping of devices and allow referencing all devices by a single name. This simplifies the process of assigning multiple devices to a subject.

Note: A device class may contain an arbitrary number of devices, including zero.

See line 603 in listing 8.1 for a device class example.

Attributes

Name	Type	Use
name	nameType	required Device class name.

Structure



7.1.45 physRefType

Reference to physical device or physical device resource.

Attributes

Name	Type	Use
physical	nameType	required Physical resource name (device or resource sub-element).

7.1.46 kernelDiagType

The debug build Muen SK can be instructed to output debugging information during runtime. The platform diagnostics device specifies which device the kernel is to use for this purpose.

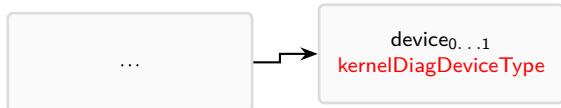
The presence of this device and the necessary resources are checked by the validator tool.

See line 623 in listing 8.1 for an example platform diagnostic device configuration.

Attributes

Name	Type	Use
type	kernelDiagKindType	required Specifies the type of diagnostics device to use.

Structure



7.1.47 kernelDiagDeviceType

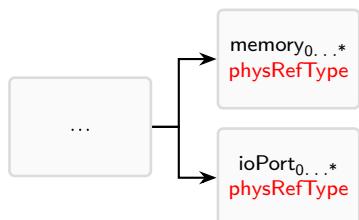
Reference to physical device for uart and vga diagnostic device type.

If an UART device is referenced via type uart, an I/O port resource must be provided. If a VGA device is referenced via type vga, a memory resource must be provided (both checked via validator).

Attributes

Name	Type	Use
physical	nameType	required Name of physical device to use for kernel diagnostics output.

Structure



7.1.48 kernelDiagKindType

Base: xs:string

Type of diagnostics device. While none disables kernel diagnostics output, uart specifies an Universal Asynchronous Receiver-Transmitter serial device. huart is a High-Speed UART with memory mapped I/O.

vga outputs the kernel diagnostics information to a VGA console, which is mainly useful for initial bring-up of a new hardware platform with no UART device.

Restrictions

values:

- none
- uart
- huart
- vga

7.1.49 memRegionsType

This section declares all physical memory regions (RAM) and thus the physical memory layout of the system. Regions declared in this section can be assigned to subjects and device domains.

Memory regions are defined by the following attributes:

- Name
- Caching type
- Size
- Physical address*

- Alignment*
- Memory type*

Attributes with an asterisk are optional. While alignment and memory type are set to a default value if not specified, the physical address is filled in by the allocator tool, which allocates all memory regions and finalizes the physical memory layout.

Additionally, the content of a region can be declared as backed by a file or filled with a pattern.

Note: The caching type is an attribute of the physical memory region by design to avoid inconsistent typing, even though the Intel Page Attribute Table (PAT) mechanism allows to set it for each memory mapping, see Intel SDM Vol. 3A, "11.12.4 Programming the PAT".

See line 637 in listing 8.1 for an example memory region section.

Structure



7.1.50 memoryType

Base: physicalMemBaseType < memoryBaseType

The memoryType specifies a physical memory region by name, size and caching.

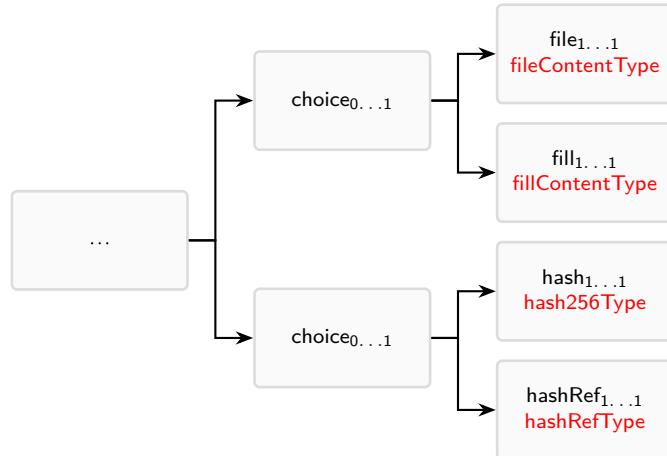
If no explicit physical address is specified for the region, the mucfgalloc tool will allocate a free one in usable memory, honoring the optional alignment attribute.

If no explicit alignment attribute is specified, it is set to 16#1000# by the expander. If no explicit type attribute is specified, it is set to subject by the expander.

Attributes

Name	Type	Use
size	memorySizeType	required Size of region. Must be a multiple of page size (4K). Enforced by validator.
name	nameType	required Name of region.
caching	cachingType	required Caching type to use for memory region.
type	subjectMemoryKindType	optional Optional subject memory type.
alignment	alignmentType	optional Alignment the physical address of the memory region must honor (checked by the validator tool).
physicalAddress	word64Type	optional Physical address of memory region.

Structure



7.1.51 subjectMemoryKindType

Base: memoryKindType < xs:string

Subject memory type to categorize memory assigned to a subject. The validator tool checks that a subject only maps memory regions of types outlined in this section (6.12).

Also used by build tools to lookup certain elements by type. For example, the mugenzp tool looks for subject memory of type subject_zeropage to process all Linux zero-pages in the policy.

The following memory types are currently supported:

- **subject**
Generic subject memory, used e.g. for RAM regions of VM subjects. The mugenzp tool used for Linux VMs (5.2.9) exports such regions as E820_RAM in the ZP E820 memory map.
- **subject_info**
Subject info (sinfo) region provided to all subjects. The sinfo region is used to query information about the execution environment. The file backing of this region is created by the mugensinfo tool (5.2.9).
- **subject_state**
Subject execution state. Mapped into the SK kernel executing the given subject, kernels running on other CPUs have no access. Accessible by subject monitors running on the same CPU if specified in the policy. Validator enforces that each subject has an associated subject_state region and that it is mapped at the expected virtual address in the executing kernel (6.12).
- **subject_binary**
Subject executable as a whole or separate subject executable regions (text, rodata, data, bss, stack) with access rights (writable/executable). The mucbinsplit tool automatically creates a component provides section with separate binary regions and associated backing files from a component binary (5.3.4).
- **subject_channel**
Physical memory region used as shared channel between two subjects. The expander tool transforms channels in system policy source format to memory regions with this type in system policy format A/B, as described in section 7.1.69.
- **subject_crash_audit**
Memory region used by crash audit facility to store system crash information into slots, see [2]. This information is preserved after a crash by performing a system warm start. Validators enforce that
 - Region is present and uncached, 6.8
 - Region does not overlap with image, 6.8

- Kernel mappings are present and correct, [6.7](#)
- No subject has write access to this region, [6.12](#)
- `subject_initrd`
Physical memory of this type designates an *initial ramdisk*. This memory type is mostly used by Linux VMs. If multiple initrd regions are mapped into a subject, they must be adjacent ([6.12](#)).
The `mugenzp` tool ([5.2.9](#)) extracts the virtual address and size of a subject-mapped region of this memory type and stores the values in the generated Linux zero-page (ZP) backing file.
- `subject_bios`
Indicates to subjects that the memory region is reserved for BIOS/firmware and must not be used as regular RAM.
- `subject_acpi_*`
Indicates to subjects that the memory region contains an ACPI table. See the ACPI specification for more information about RSDP, XSDT, FADT and DSDT ACPI tables. The `mugenzp` tool ([5.2.9](#)) exports such regions as `E820_ACPI` in the ZP `E820` memory map.
- `subject_zeropage`
Indicates to Linux subjects that the memory region contains a zero-page. See the Linux kernel Zero Page documentation for more information.
- `subject_solo5_boot_info`
Indicates to a VM running Solo5/Mirage that the memory region contains a boot info structure. The file-backing of such a region may be created using the `mugensolo5` tool ([5.2.9](#)).
- `subject_device`
Designates a memory region which is allowed to be added to a subject and a device domain. The difference to the `subject` memory type is that the region is not exported as `E820_RAM` but `E820_RESERVED` to Linux subjects. Therefore, such a region is useful to implement custom drivers without interference from Linux DMA zone handling.
- `subject_timed_event`
Region designates a subject timed event page, as described in [\[1\]](#). The expander tool creates a physical memory region for each subject and maps it into the associated subject and the SK kernel executing this subject.

Restrictions

values:

- `subject`
- `subject_info`
- `subject_state`
- `subject_binary`
- `subject_channel`
- `subject_crash_audit`
- `subject_initrd`
- `subject_bios`
- `subject_acpi_rsdp`
- `subject_acpi_xsdt`
- `subject_acpi_fadt`

- subject_acpi_dsdt
- subject_zeropage
- subject_solo5_boot_info
- subject_device
- subject_timed_event

7.1.52 alignmentType

Base: word64Type < xs:string

Memory alignment constraint for memory region. Taken into account by the allocator tool and checked by the validator.

Restrictions

values:

- 16#1000#
- 16#0020_0000#
- 16#4000_0000#

7.1.53 fileContentType

The `file` child element designates a file-backed memory region.

The `filename` attribute specifies the name of the file to use as content for the physical memory region, the `offset` attribute is none by default but can be customized to include a partial file.

See line 717 in listing 8.1 for a file-backed memory region example. The following checks on the file content are performed.

- If `offset` is none, the size of the file must be less than the memory region size.
- If `offset` is not none, the offset must be less than the file size. The file size is *not* checked but the memory region size is used as upper bound.

Attributes

Name	Type	Use
filename	xs:string	required Filename of file to (partially) include. Note that the actual file processed by the toolchain also depends on the working directory passed as command line option to the specific tool.
offset	optionalOffsetType	required Read file offset in bytes.

7.1.54 optionalOffsetType

Optional file offset value in bytes.

Restrictions

Union of

- word64Type
- noneType

7.1.55 fillContentType

The `fill` element designates a memory region which is initialized with the given pattern.

See line 671 in listing 8.1 for a file-backed memory region example.

Attributes

Name	Type	Use
pattern	byteType	required Fill pattern (hex).

7.1.56 hash256Type

The hash child element of a memory region designates a 256-bit hash over the memory content.

The Mucfgmemhashes tool in the Muen toolchain generates such a hash-sum for every content-backed memory region in a given policy.

Attributes

Name	Type	Use
value	optionalHashType	required 256-Bits message digest over file-backed memory content.

7.1.57 optionalHashType

Allows the specification of a hash digest or none.

Restrictions

Union of

- hash256DigestType
- noneType

7.1.58 hashRefType

The optional hashRef child element of a physical memory region instructs the Mucfgmemhashes tool to copy the hash element of the referenced memory region after message digest generation.

From an abstract point of view, the hashRef element is a way to link multiple memory regions by declaring that the hash of the content is the same. This concept is e.g. used by the subject loader mechanism to restore writable memory regions to their initial state.

Attributes

Name	Type	Use
memory	nameType	required Name of referenced physical memory region.

7.1.59 deviceDomainsType

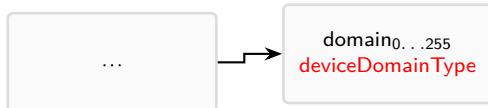
The physical memory accessible by PCI devices is specified by so called device domains. Such domains define memory mappings of physical memory regions for one or multiple devices. Device references select a subset of hardware devices provided by the hardware/platform. Devices may be referenced by device name, alias or device class.

Device references can optionally set the mapReservedMemory attribute so RMRR regions referenced by the device are also mapped into the device domain.

Device domains are isolated from each other by the use of Intel VT-d.

See line 733 in listing 8.1 for a device domain example.

Structure



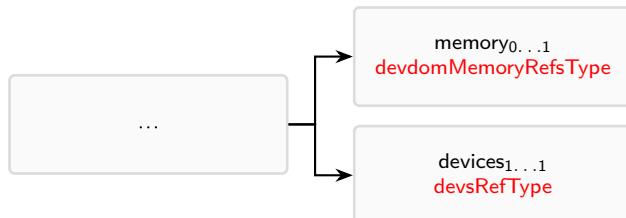
7.1.60 deviceDomainType

A device domain allows referenced devices access to the specified memory regions. It also provides handling for reserved memory region reporting (RMRR), see Intel VT-d Specification, "8.4 Reserved Memory Region Reporting Structure".

Attributes

Name	Type	Use
name	nameType	required Name of the device domain.

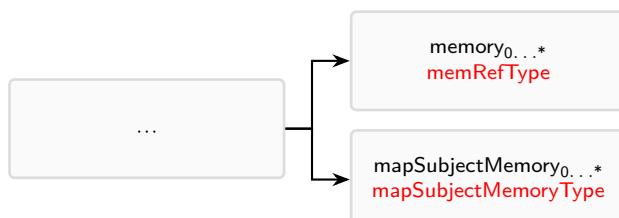
Structure



7.1.61 devdomMemoryRefsType

List of physical memory region references and optional map subject memory elements.

Structure



7.1.62 memRefType

A memory element maps a physical memory region into the address space of a device domain or subject entity. The region will be accessible to the entity at the specified virtualAddress with permissions defined by the executable and writable attributes.

See line 758 in listing 8.1 for an example of such a mapping.

Attributes

Name	Type	Use
virtualAddress	word64Type	required Address in entity address space where the physical memory region is mapped.
physical	nameType	required Name of referenced physical memory region.
logical	nameType	required Logical name of mapping.
writable	booleanType	required Defines if the mapped memory is writable.

```
(continuation)
Name      Type      Use
executable booleanType required
Defines if the memory region contents are executable by the processor.
```

7.1.63 mapSubjectMemoryType

This element instructs the expander to map memory regions of the specified subject into the device domain. Only regions which are writable and of type *Subject* and *Subject_Initrd* are mapped.

Attributes

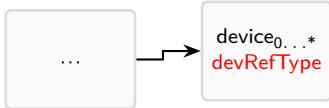
Name	Type	Use
subject	nameType	required
	Name of the subject.	

virtualAddressOffset	word64Type	optional
	Optional offset value. If this attribute is specified, the given value will be added to the virtualAddress value of a mapped memory region.	

7.1.64 devsRefType

Device domain device references.

Structure



7.1.65 devRefType

Device domain device reference. Referenced devices gain access to memory regions of device domain.

Attributes

Name	Type	Use
logical	nameType	required
	Logical name in this context.	
physical	nameType	required
	Physical device or device alias to include in given device domain.	
mapReservedMemory	booleanType	optional
	Whether to automatically map RMRR memory associated with device.	

7.1.66 eventsType

Events are an activity caused by a subject (source) that impacts a second subject (target) or is directed at the kernel. Events are declared globally and have a unique name to be unambiguous. An event must have a single source and one target.

Subjects can use events to either deliver an interrupt, hand over execution to or reset the state of a target subject. The first kind of event provides a basic notification mechanism and enables the implementation of event-driven services. The second type facilitates suspension of execution of the source subject and switching to the target. Such a construct is used to pass the thread of execution on to a different subject, e.g. invocation of a debugger subject if an error occurs in the source subject. The third kind is used to facilitate the restart of subjects.

An event can also have the same source and target, which is called *self* event. Such events are useful to implement para-virtualized timers in VM subjects for example.

Kernel events are special in that they are targeted at the kernel. The currently supported events are system reboot and shutdown.

For documentation about linking physical events to source and target subjects, see section [7.1.130](#).

See line [774](#) in listing [8.1](#) for an example events section.

Structure



7.1.67 eventType

The `eventType` specifies an event by name and mode.

The following event modes are currently supported:

- `asap`

The `asap` event is an abstraction to state that the event should be delivered as soon as possible, depending on the CPU of the target subject. If the target runs on another CPU core, this mode is expanded to mode *ipi*, which is only available in policy formats A and B, instructing the kernel to preempt the kernel running the target subject and inject the event immediately. If the target subject runs on the same core as the source subject, the mode is expanded to mode *async*.

- `async`

`Async` events trigger no preemption at the target subject. The event is marked as pending in the target subject's pending event table and inserted on the next VM exit/entry cycle of the target subject.

- `self`

An event can also have the same source and target, which is called a `self` event. Such events are useful to implement para-virtualized timers in VM subjects for example. A subject sends itself a delayed event, using the timed event mechanism. Note that a `self` event must always have a target action assigned, which is checked by the validator.

- `switch`

The `switch` mode facilitates suspension of execution of the source subject and switching to the target. This can only happen between subjects running on the same core. Such a construct is used to pass the thread of execution on to a different subject, e.g. invocation of a debugger subject if an error occurs in the source subject. It is called *handover* or *handover event*.

- `kernel`

These kinds of events are directed at the kernel and thus only specify a source since the target is the kernel. They are used to enable specific subjects to unmask level-triggered IRQs and trigger a system reboot, poweroff or explicit panic (crash audit slot allocation and reboot).

See line [798](#) in listing [8.1](#) for an example global event declaration.

Attributes

Name	Type	Use
<code>name</code>	<code>nameType</code>	required Name of the event.
<code>mode</code>	<code>eventModeType</code>	required Mode of the event.

7.1.68 eventModeType

Base: `xs:string`

Event mode.

See [7.1.67](#) for details about the supported event modes.

Restrictions

values:

- asap
- async
- self
- switch
- kernel

7.1.69 channelsType

Inter-subject communication is specified by so called channels. These channels represent directed information flows since they have a single writer and possibly multiple readers. Optionally a channel can have an associated notification event (doorbell interrupt).

Channels are declared globally and have an unique name to be unambiguous.

Note that channels are a policy source format abstraction. The toolchain resolves this concept into memory regions and events as well as the appropriate subject mappings.

For documentation about linking physical channels to subjects see section [7.1.138](#). For documentation about declaring requested channels in components see section [7.1.102](#). For information how to map a physical channel with a logical component channel at subject level, see section [7.1.145](#).

See line [864](#) in listing [8.1](#) for an example channel section.

Structure



7.1.70 channelType

The `channel` element declares a physical channel.

Besides the `name` and `size` of the channel, the optional `hasEvent` attribute can be set to declare that the given channel requests an associated event. The expander tool will then automatically create a global event of the requested event type.

See line [878](#) in listing [8.1](#) for an example channel declaration.

Attributes

Name	Type	Use
<code>name</code>	<code>nameType</code>	required
Channel name.		
<code>size</code>	<code>memorySizeType</code>	required
Size of the channel in bytes.	Must be a multiple of page size (4K). Enforced by validator.	
<code>hasEvent</code>	<code>eventModeType</code>	optional
Associated event type (if any).		

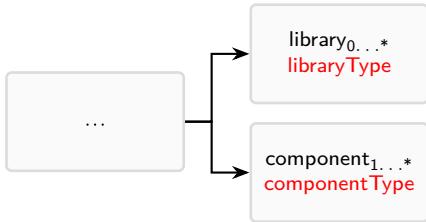
7.1.71 componentsType

The `components` element holds a list of components and component libraries.

Note that components are a policy source format abstraction. The toolchain resolves this concept into subjects by adding the appropriate memory regions, events and devices.

See line [913](#) in listing [8.1](#) for an example components section.

Structure



7.1.72 libraryType

A component library is a specialized component specification which is used to share common resources required for library code to operate. Component libraries can be included by multiple components in order to share functionality. An example is a logging service provided by a dedicated component, whereas the logging client is provided as a library with a shared memory channel for the actual log messages.

A component specification declares library dependencies to request the library resources from the system through the inclusion of the library specification in the depends section. This way components inherit the resources of libraries.

On the source code level, a library is included by mechanisms provided by the respective programming language. Note that the component library code is *not* shared between components but lives in the isolated execution environment of a subject instantiating the component (i.e. statically linked libraries).

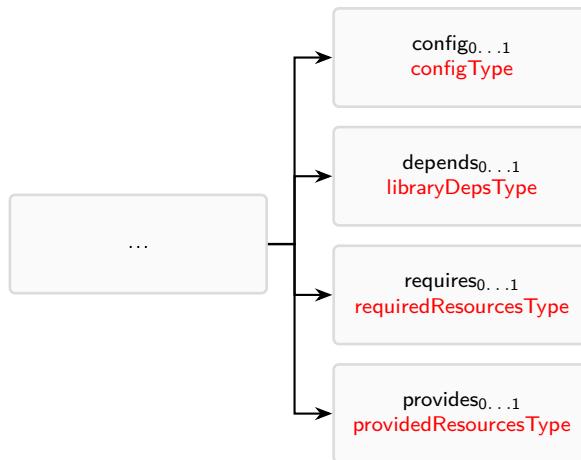
Libraries can request the same resources as ordinary components. A subject instantiating the component must also map the resources requested by libraries the component depends on.

See line 922 in listing 8.1 for example library specifications.

Attributes

Name	Type	Use
name	nameType	required Component/library name.

Structure



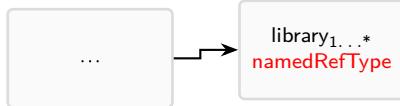
7.1.73 libraryDepsType

Components and libraries are allowed to declare dependencies to other libraries. All resources required by the included library are merged with the ones specified by the component or library. Libraries can depend on other libraries.

A subject realizing this component must correctly map all component and library resource requirements to physical resources in order to fulfill the expectations.

See line 1077 in listing 8.1 for an example dependency section.

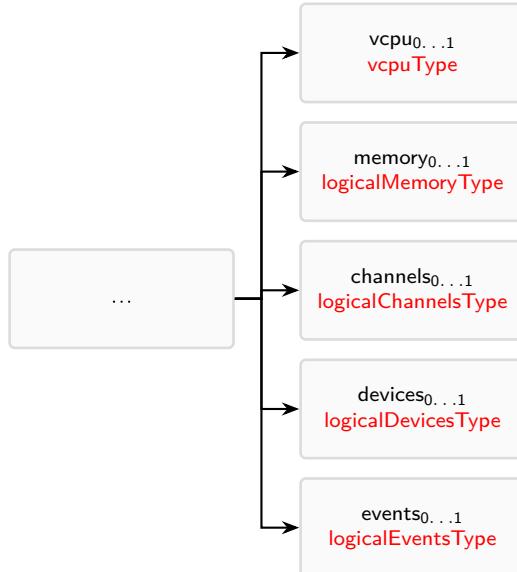
Structure



7.1.74 requiredResourcesType

Declaration of resources a component or library requires to operate.

Structure

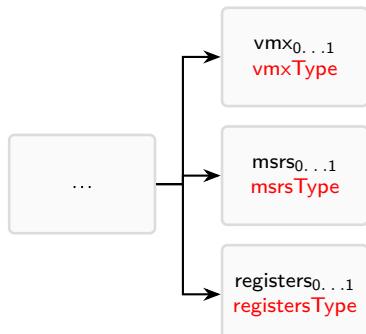


7.1.75 vcpuType

The vcpu element controls the execution behavior of the virtual CPU (vCPU). A default vCPU profile is selected by the component profile, but CPU execution settings can be customized both at component and subject level.

See line 1027 in listing 8.1 for an example on how to customize a vCPU profile.

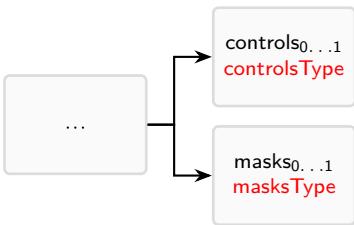
Structure



7.1.76 vmxType

Controls Intel VMX vCPU settings.

Structure

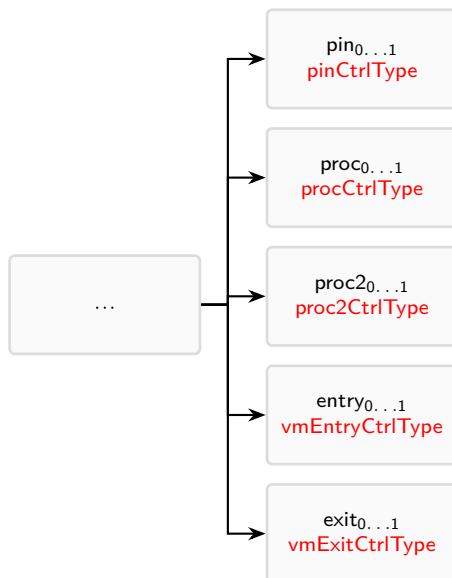


7.1.77 controlsType

Configures the following Intel VMX settings:

- Pin-Based VM-Execution Controls
- Primary Processor-Based VM-Execution Controls
- Secondary Processor-Based VM-Execution Controls
- VM-Entry Controls
- VM-Exit Controls

Structure



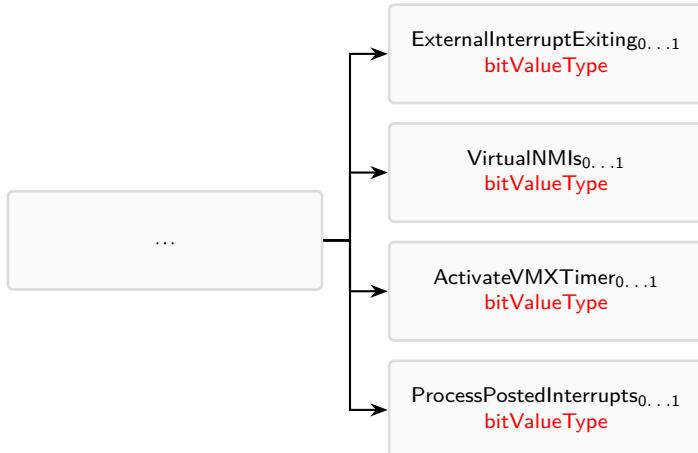
7.1.78 pinCtrlType

Configures Intel VMX pin-based VM-execution controls. These controls constitute a 32-bit vector that governs the handling of asynchronous events (for example: interrupts) while running in VMX non-root mode.

See Intel SDM Vol. 3C, "24.6.1 Pin-Based VM-Execution Controls" for more details and the meaning of the different bit-fields.

 Deviating from the settings provided by the component vCPU profile might result in unexpected system behavior. A system integrator changing any of these values must have a thorough understanding of both the runtime behavior of the Muen SK and the Intel VT-x/VT-d architecture. The mucfgvalidate tool checks that requirements for safe execution of Muen are met, i.e. invalid settings are detected and a meaningful error message is presented.

Structure



7.1.79 bitValueType

Base: xs:nonNegativeInteger

The value of one bit, either 1 (True) 0 (False).

Restrictions

$0 \leq 1$

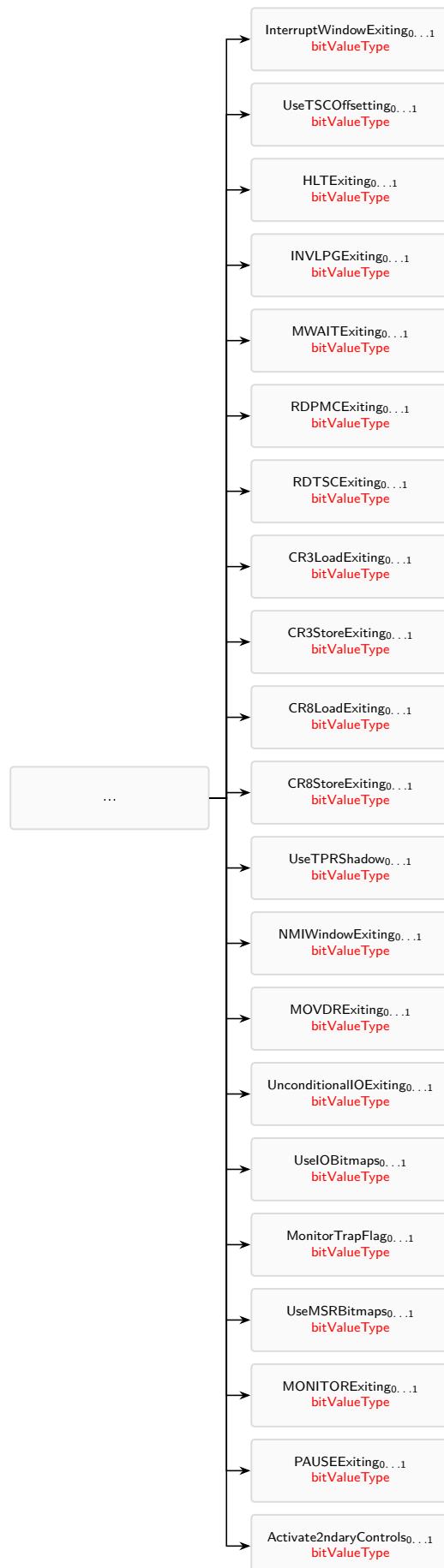
7.1.80 procCtrlType

The processor-based VM-execution controls constitute two 32-bit vectors that govern the handling of synchronous events, mainly those caused by the execution of specific instructions. These are the *primary processor-based* VM-execution controls and the *secondary processor-based* VM-execution controls.

The proc element configures the primary processor-based VM-execution controls, see Intel SDM Vol. 3C, "24.6.2 Processor-Based VM-Execution Controls" for more details and the meaning of the different bit-fields.

 Deviating from the settings provided by the component vCPU profile might result in unexpected system behavior. A system integrator changing any of these values must have a thorough understanding of both the runtime behavior of the Muen SK and the Intel VT-x/VT-d architecture. The mucfgvalidate tool checks that requirements for safe execution of Muen are met, i.e. invalid settings are detected and a meaningful error message is presented.

Structure



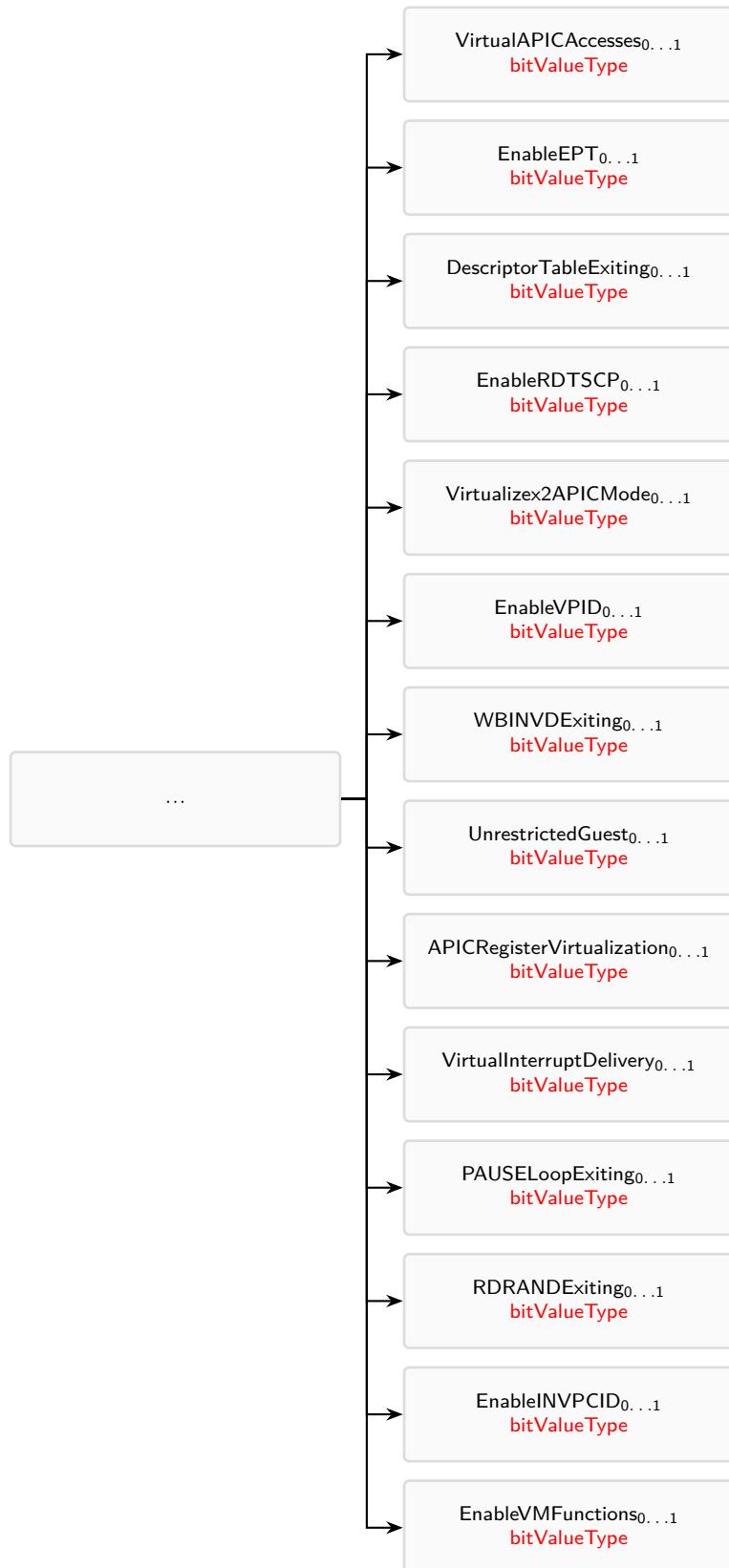
7.1.81 proc2CtrlType

The processor-based VM-execution controls constitute two 32-bit vectors that govern the handling of synchronous events, mainly those caused by the execution of specific instructions. These are the *primary processor-based* VM-execution controls and the *secondary processor-based* VM-execution controls.

The `proc2` element configures the secondary processor-based VM-execution controls, see Intel SDM Vol. 3C, "24.6.2 Processor-Based VM-Execution Controls" for more details and the meaning of the different bit-fields.

 Deviating from the settings provided by the component vCPU profile might result in unexpected system behavior. A system integrator changing any of these values must have a thorough understanding of both the runtime behavior of the Muen SK and the Intel VT-x/VT-d architecture. The `mucfgvalidate` tool checks that requirements for safe execution of Muen are met, i.e. invalid settings are detected and a meaningful error message is presented.

Structure



7.1.82 `vmEntryCtrlType`

Configures Intel VMX VM-entry controls. These controls constitute a 32-bit vector that governs the basic operation of VM entries.

See Intel SDM Vol. 3C, "24.8.1 VM-Entry Controls" for more details and the meaning of the different bit-fields.

 Deviating from the settings provided by the component vCPU profile might result in unexpected system behavior. A system integrator changing any of these values must have a thorough understanding of both the runtime behavior of the Muen SK and the Intel VT-x/VT-d architecture. The mucfgvalidate tool checks that requirements for safe execution of Muen are met, i.e. invalid settings are detected and a meaningful error message is presented.

Structure



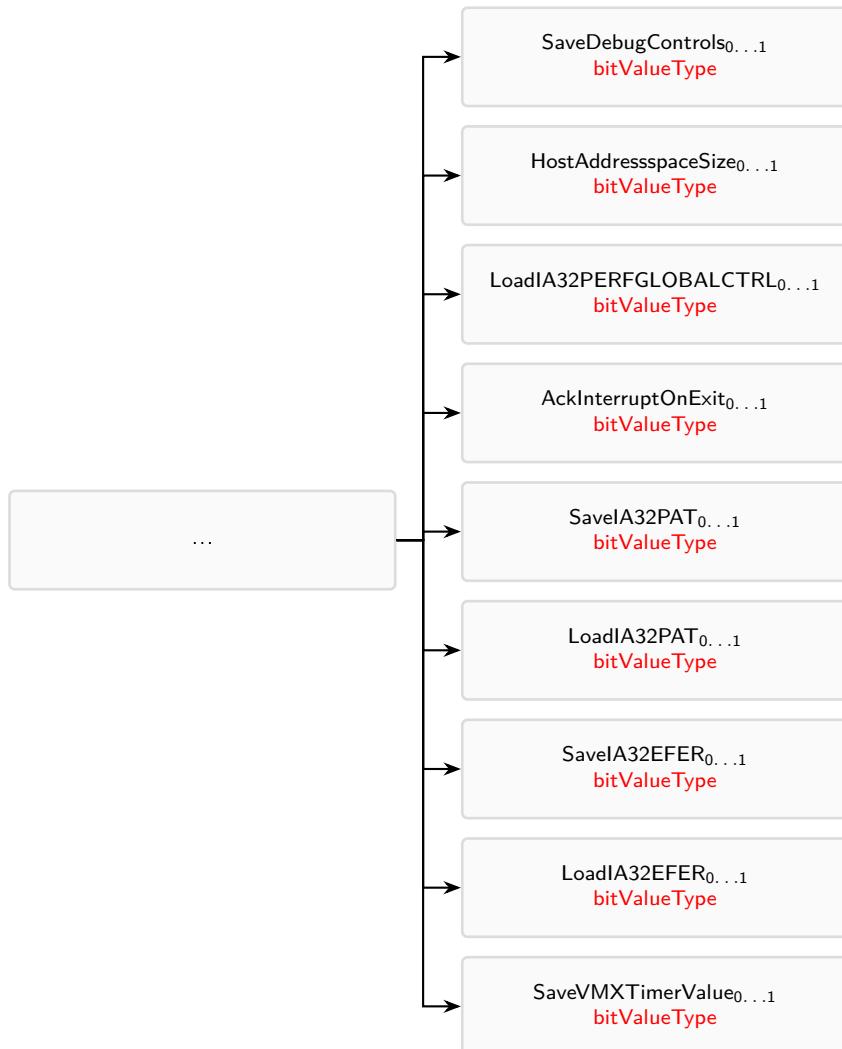
7.1.83 vmExitCtrlType

Configures Intel VMX VM-exit controls. These controls constitute a 32-bit vector that governs the basic operation of VM exits.

See Intel SDM Vol. 3C, "24.7.1 VM-Exit Controls" for more details and the meaning of the different bit-fields.

 Deviating from the settings provided by the component vCPU profile might result in unexpected system behavior. A system integrator changing any of these values must have a thorough understanding of both the runtime behavior of the Muen SK and the Intel VT-x/VT-d architecture. The mucfgvalidate tool checks that requirements for safe execution of Muen are met, i.e. invalid settings are detected and a meaningful error message is presented.

Structure



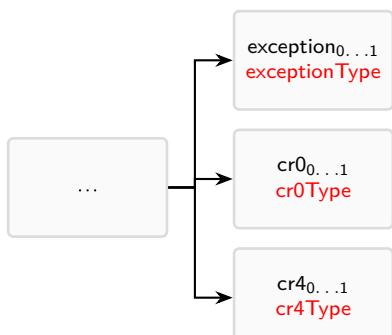
7.1.84 masksType

The `masks` element configures the Intel VMX CR0/CR4 guest/host masks and the guest/host exception bitmap.

In general, bits set to 1 in a guest/host mask correspond to bits *owned* by the host, causing a VM exit if the associated event occurs.

Reading from host owned bits in CR0/CR4 does not result in a VM exit but the value of the CR0/CR4 read shadow is returned instead (see Intel SDM Vol. 3C, "24.6.6 Guest/Host Masks and Read Shadows for CR0 and CR4").

Structure

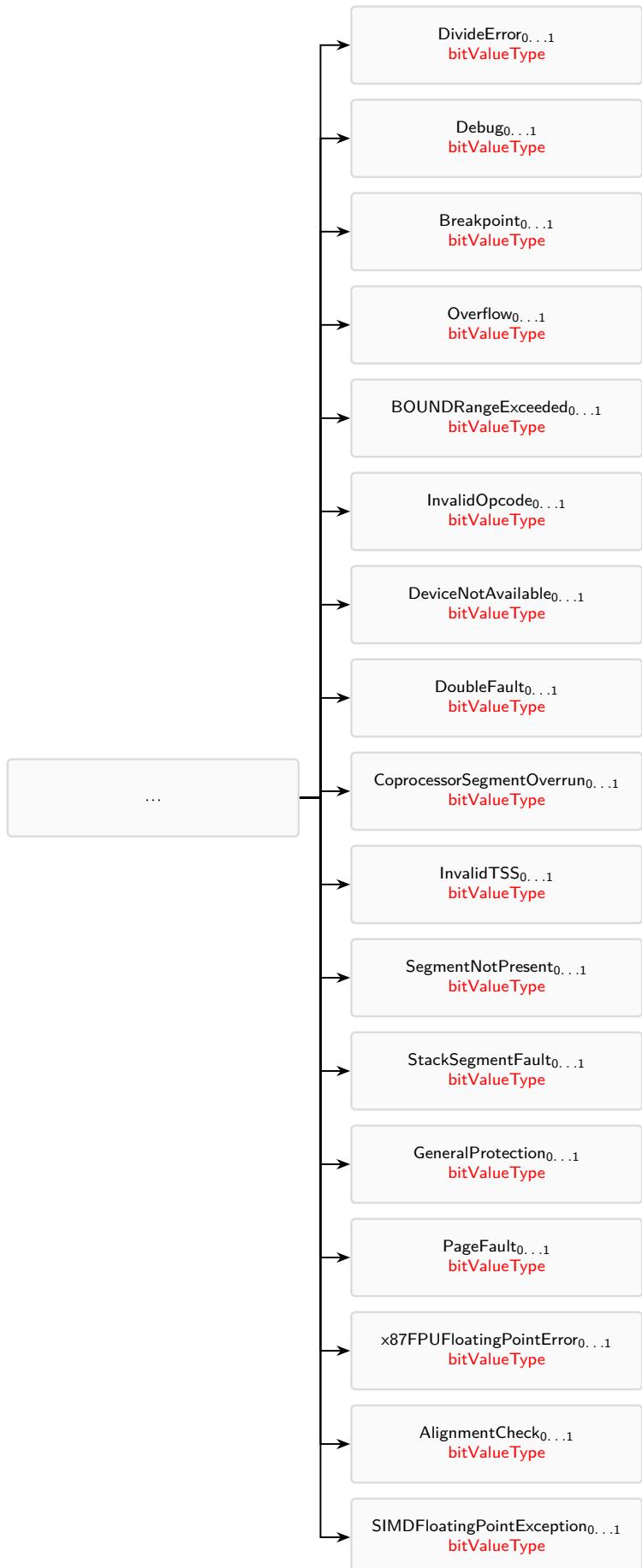


7.1.85 exceptionType

Configures Intel VMX exception bitmap. The exception bitmap is a 32-bit field that contains one bit for each exception. When an exception occurs, its vector is used to select a bit in this field. If the bit is 1, the exception causes a VM exit. If the bit is 0, the exception is delivered normally through the IDT, using the descriptor corresponding to the exceptions vector.

See Intel SDM Vol. 3C, "24.6.3 Exception Bitmap" for more details on the exception bitmap configuration.

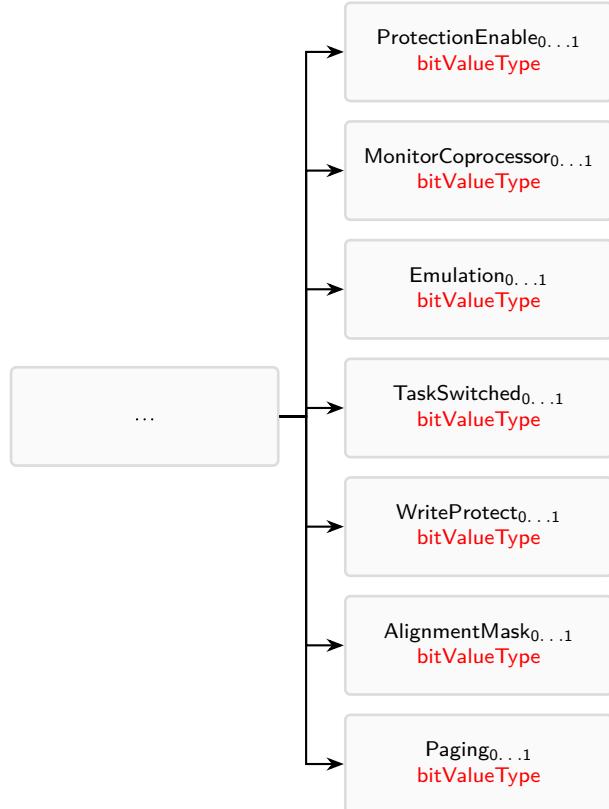
Structure



7.1.86 cr0Type

Allows to set initial values of the CR0 control register or bits in the CR0 guest/host ownership mask.

Structure



7.1.87 cr4Type

Allows to set initial values of the CR4 control register or bits in the CR4 guest/host ownership mask.

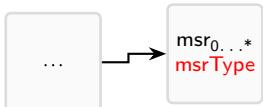
Structure



7.1.88 msrsType

List of model-specific registers (MSRs) a subject is allowed to access. The settings in this section are translated to the MSR bitmaps of the associated subject (as described by Intel SDM Vol. 3C, "24.6.9 MSR-Bitmap Address").

Structure



7.1.89 msrType

An `msr` element allows a subject direct access to the specified model-specific register (MSR).

 Deviating from the settings provided by the component vCPU profile might result in unexpected system behavior. A system integrator granting direct access to MSRs must be aware of the potential side-effects.

Attributes

Name	Type	Use
start	<code>msrAddressType</code>	required MSR start address.
end	<code>msrAddressType</code>	required MSR end address.
mode	<code>msrModeType</code>	required MSR access permissions.

7.1.90 msrAddressType

Base: xs:string

Start/end address value for MSRs in the low or high range:

- Low : `16#0000_0000# .. 16#0000_1fff#`
- High : `16#C000_0000# .. 16#C000_1fff#`

See also Intel SDM Vol. 3C, "24.6.9 MSR-Bitmap Address".

Restrictions

Pattern = `16#[([cC]000_)?[01]([0-9a-fA-F]3)#`

7.1.91 msrModeType

Base: xs:string

MSR access rights.

Restrictions

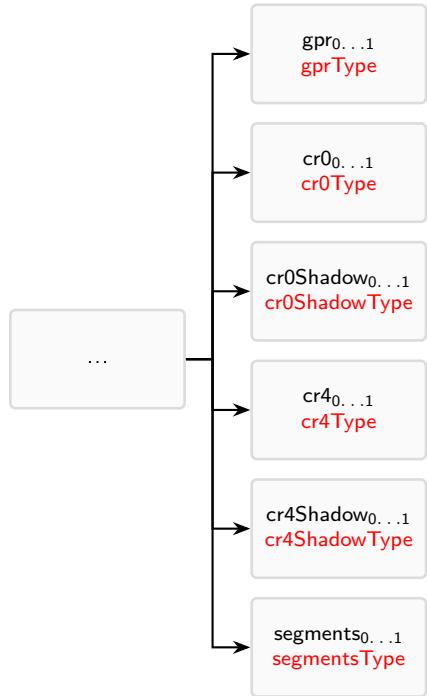
values:

- r
- w
- rw

7.1.92 registersType

The `registers` element specifies the initial value of general-purpose (GPR), CR0/CR4, CR0/CR4 read shadow and segment registers.

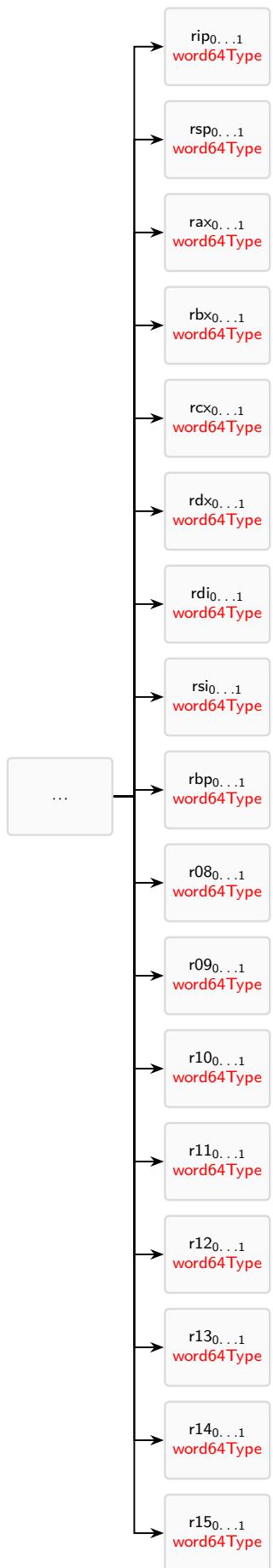
Structure



7.1.93 gprType

The `gpr` element specifies the initial values of subject general-purpose registers (GPRs).

Structure

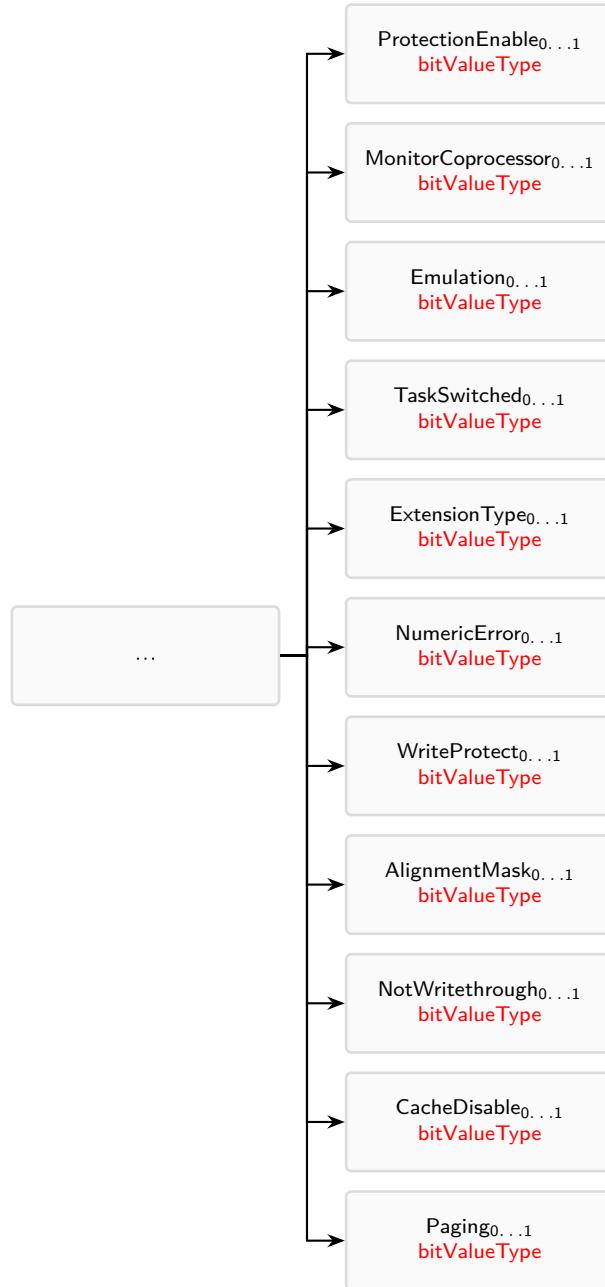


7.1.94 cr0ShadowType

Allows to set initial values of the CR0 shadow control register.

See Intel SDM Vol. 3C, "24.6.6 Guest/Host Masks and Read Shadows for CR0 and CR4" for more details on the CR0 shadow.

Structure

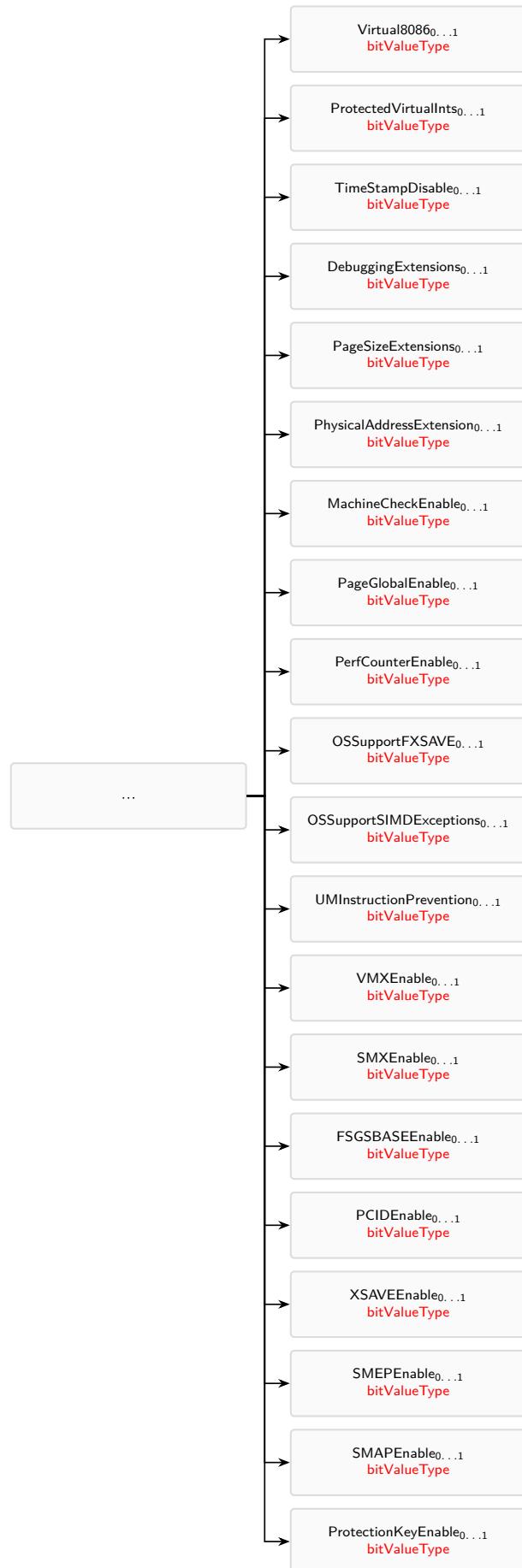


7.1.95 cr4ShadowType

Allows to set initial values of the CR4 shadow control register.

See Intel SDM Vol. 3C, "24.6.6 Guest/Host Masks and Read Shadows for CR0 and CR4" for more details on the CR0 shadow.

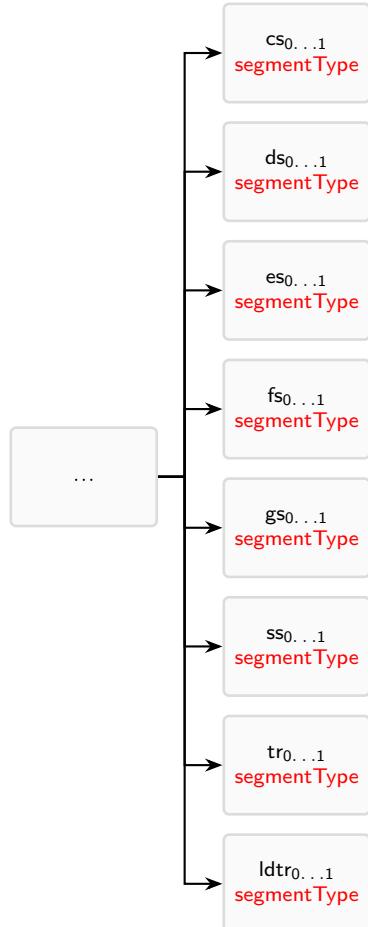
Structure



7.1.96 segmentsType

The segments element specifies the initial values of subject segment registers.

Structure



7.1.97 segmentType

Initial value of a segment register, including hidden part. See Intel SDM Vol. 3A, "3.4.3 Segment Registers" for more details on segment registers.

Attributes

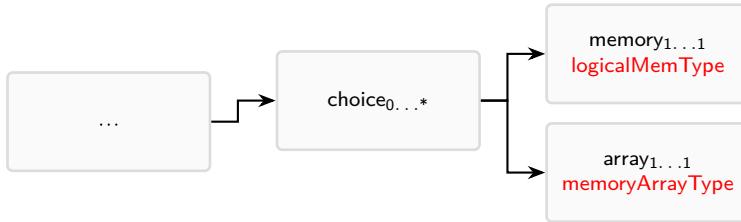
Name	Type	Use
selector	word16Type	required Segment selector value.
base	word64Type	required Segment base address.
limit	word32Type	required Segment limit.
access	word32Type	required Segment access information.

7.1.98 logicalMemoryType

In this section, components can specify expected memory mappings with given access rights and region size.

See line 948 in listing 8.1 for an example specification.

Structure



7.1.99 logicalMemType

The memory element requests a memory region with the specified `size` and permissions from the system. The region is expected to be placed at the address given via the `virtualAddress` attribute.

See line 953 in listing 8.1 for an example specification.

Attributes

Name	Type	Use
<code>size</code>	<code>word64Type</code>	required Size of memory in bytes. Must be a multiple of page size (4K).
<code>virtualAddress</code>	<code>word64Type</code>	required Expected address of memory mapping.
<code>logical</code>	<code>nameType</code>	required Logical name of mapping.
<code>writable</code>	<code>booleanType</code>	required Defines if the mapped memory is writable.
<code>executable</code>	<code>booleanType</code>	required Defines if the memory region contents are executable by the processor.

7.1.100 memoryArrayType

The memory array abstraction simplifies the declaration of consecutive memory mappings with a given base address, region size and executable and writable attributes. The child elements declare the number of expected regions.

Attributes

Name	Type	Use
<code>logical</code>	<code>nameType</code>	required Logical name of mapping.
<code>writable</code>	<code>booleanType</code>	required Defines if the mapped memory is writable.
<code>executable</code>	<code>booleanType</code>	required Defines if the memory region contents are executable by the processor.
<code>virtualAddressBase</code>	<code>word64Type</code>	required Expected address of memory mapping.
<code>elementSize</code>	<code>word64Type</code>	required Size of one array element in bytes. Must be a multiple of page size (4K).

Structure



7.1.101 arrayEntryType

Array entries specify the number of array elements and assign a logical name to each element.

See line 1114 in listing 8.1 for an example array entry declaration.

Attributes

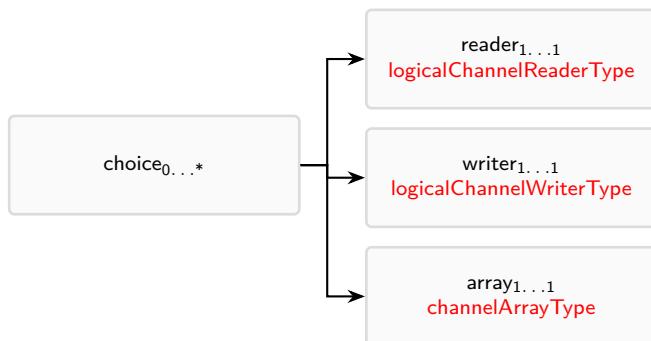
Name	Type	Use
logical	nameType	required Logical name of array entry.

7.1.102 logicalChannelsType

Components and libraries use the channels sub-section of requires to specify expected communication channels.

See line 969 in listing 8.1 for an example specification.

Structure



7.1.103 logicalChannelReaderType

The reader element requests a read-only channel of the specified size, address and optional notification vector.

See line 987 in listing 8.1 for an example channel reader specification.

Attributes

Name	Type	Use
logical	nameType	required Logical name of reader channel.
virtualAddress	word64Type	required Expected address of channel memory mapping.
size	word64Type	required Expected size of channel. Must be a multiple of page size (4K).
vector	vectorType	optional Notification vector.

7.1.104 vectorType

Base: xs:nonNegativeInteger
Vector number.

Restrictions

value ≤ 255

7.1.105 logicalChannelWriterType

The writer element requests a channel with write permissions of the specified size, address and optional notification event number. For valid event ID ranges, see vmcall group in [7.1.134](#).

See line [974](#) in listing [8.1](#) for an example channel writer specification.

Attributes

Name	Type	Use
logical	nameType	required Logical name of writer channel.
virtualAddress	word64Type	required Expected address of channel memory mapping.
size	word64Type	required Expected size of channel. Must be a multiple of page size (4K).
event	xs:nonNegativeInteger	optional Notification event number.

7.1.106 channelArrayType

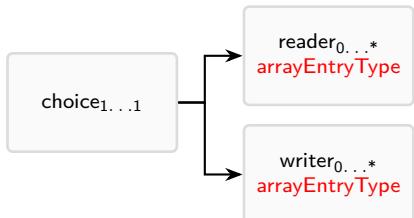
The channel array abstraction simplifies the declaration of consecutive channel mappings with a given base address, channel size and optional event/vector bases. The child elements declare the number of expected channels and either the reader or writer role.

See line [1107](#) in listing [8.1](#) for an example specification.

Attributes

Name	Type	Use
logical	nameType	required Logical channel array name.
eventBase	xs:nonNegativeInteger	optional The eventBase attribute specifies the event number of the first element in the array. This number is incremented for all further elements in the array (eventBase + 1). For valid event ID ranges see vmcall group in 7.1.134 . Note that this attribute is only taken into consideration for a writer array.
vectorBase	vectorType	optional The vectorBase attribute specifies the vector number of the first element in the array. This number is incremented for all further elements in the array (vectorBase + 1). Note that this attribute is only taken into consideration for a reader array.
virtualAddressBase	word64Type	required Expected address of memory mapping.
elementSize	word64Type	required Size of one array element in bytes. Must be a multiple of page size (4K).

Structure

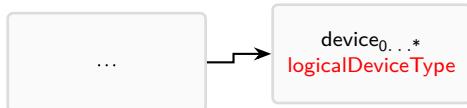


7.1.107 logicalDevicesType

The devices sub-section of the requires section is used to specify expected devices with their associated resources.

See line [1127](#) in listing [8.1](#) for an example specification.

Structure



7.1.108 logicalDeviceType

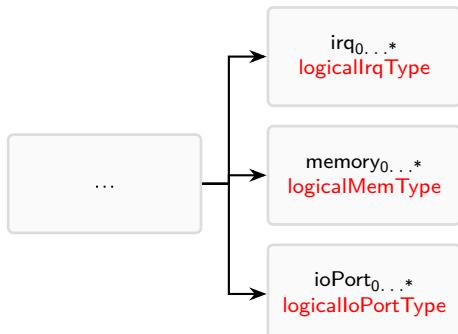
A device element specifies an expected logical device with its resources. Possible resources are irq, memory and ioPort.

See line 1132 in listing 8.1 for an example specification.

Attributes

Name	Type	Use
logical	nameType	required Logical device name.

Structure



7.1.109 logicalIRQType

An irq element of a logical device reference requests an IRQ with given number from the system policy. The specified number will be injected when the device requires attention for the associated logical function.

See line 1137 in listing 8.1 for an example IRQ reference.

Attributes

Name	Type	Use
logical	nameType	required Logical name of IRQ resource.
vector	vectorType	required Expected IRQ number.

Structure



7.1.110 logicalMSIIRQType

The presence of msi child elements of an irq device resource specifies that the component expects the device to be operated in MSI mode. The number of elements defines the expected MSI vector number count to be provided by the referenced device.

Attributes

Name	Type	Use
logical	nameType	required Name of MSI resource.

7.1.111 logicalIoPortType

The ioPort element requests a device I/O port resource with given range start .. end from the system.

See line 1296 in listing 8.1 for an example I/O port reference.

Attributes

Name	Type	Use
logical	nameType	required Logical I/O port name.
start	word16Type	required I/O port start address.
end	word16Type	required I/O port end address.

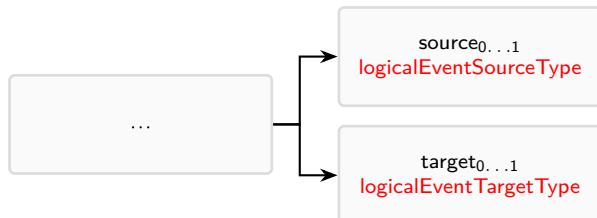
7.1.112 logicalEventsType

The events sub-section of the requires section is used to specify expected events with optional event actions.

A component can specify both source as well as target events.

See line 1223 in listing 8.1 for an example specification.

Structure



7.1.113 logicalEventSourceType

Specifies expected source events.

Structure



7.1.114 logicalSourceEventEntryType

Base: baseLogicalEventType

An entry in the component's source event list.

Attributes

Name	Type	Use
logical	nameType	required Logical name of event.

Name	Type	Use
id	xs:nonNegativeInteger	required ID of source event. For valid ID ranges see vmcall group in 7.1.134 .

7.1.115 logicalEventTargetType

Specifies expected event targets.

Structure



7.1.116 logicalTargetEventEntryType

Base: baseLogicalEventType

An entry in the component's target event list.

Attributes

Name	Type	Use
logical	nameType	required Logical name of event.
id	xs:nonNegativeInteger	optional ID of target event entry.

7.1.117 providedResourcesType

Components usually come in the form of an executable file. To this end, the provides section specifies the memory regions of the component binary executable with their content.

From a security perspective, it is often desirable to provide the different binary section as separate memory regions with the appropriate access rights, i.e. only the text section is executable, rodata is not writable and so on.

Memory specified in this sections are expanded to mapped physical regions for each subject that instantiates this component.

Note: the Mucbinsplit tool can be used to extract these section from an ELF binary into separate files and automatically add the corresponding memory elements to the component specification.

See line 1150 in listing [8.1](#) for an example provides section.

Structure



7.1.118 providedMemType

Base: memoryBaseType

A memory element in the provides section declares memory region provided by the component. Mostly used to provide (a part) of the component binary.

See line 1168 in listing [8.1](#) for an example specification.

Attributes

Name	Type	Use
size	memorySizeType	required Size of region. Must be a multiple of page size (4K). Enforced by validator.

```
(continuation)
Name          Type           Use
virtualAddress word64Type    required
  Virtual address in component address space.

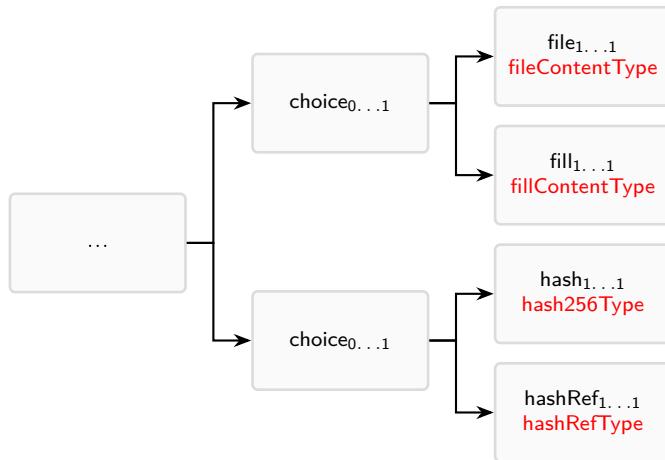
type          subjectMemoryKindType optional
  Memory type (e.g. subject_binary).

logical        nameType        required
  Logical name of mapping.

writable       booleanType     required
  Defines if the mapped memory is writable.

executable     booleanType     required
  Defines if the memory region contents are executable by the processor.
```

Structure



7.1.119 componentType

Base: libraryType

A component is a piece of software which shall be executed by the SK. Components represent the building blocks of a component-based system and can be regarded as templates for executable entities instantiated by subjects.

The specification of a component declares the *binary program* by means of (file-backed memory) regions. It also specifies the component's view of the expected execution environment. A component may request the following resources from the system:

- Logical channels
- Logical memory regions
- Logical devices
- Logical events

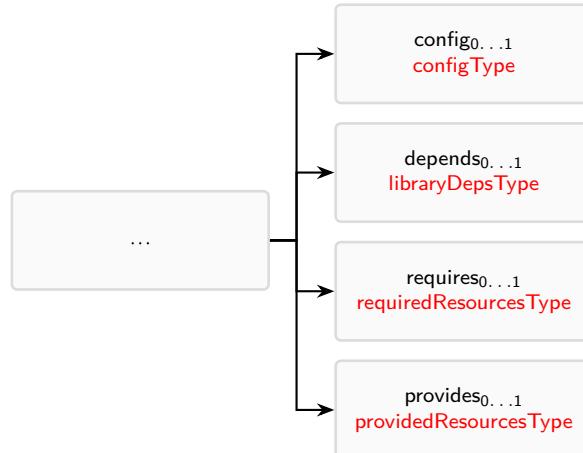
Components are identified by name and specify a profile. The profile controls the settings of the virtual CPU (vCPU).

See line 1047 in listing 8.1 for an example component.

Attributes

Name	Type	Use
name	nameType	required
	Component/library name.	
profile	componentProfileType	required
	Component profile.	

Structure



7.1.120 componentProfileType

Base: xs:string

The component profile defines default vCPU settings and triggers profile specific actions in the expander tool. The following actions are performed for the 'linux' profile.

- Add Linux zero-page (ZP, generated by Mugenzp)
- Add ACPI table regions (generated by Mugenacpi)
- Append sinfo address to boot parameters (muen_sinfo)
- Add dummy legacy BIOS regions (start address 16000c_0000)
- Invalidate guest state of Linux SMP emulation sibling subjects

Restrictions

values:

- native
- vm
- linux

7.1.121 subjectsType

The subjects element holds a list of subjects.

See line 1848 in listing 8.1 for an example subjects section.

Structure



7.1.122 subjectType

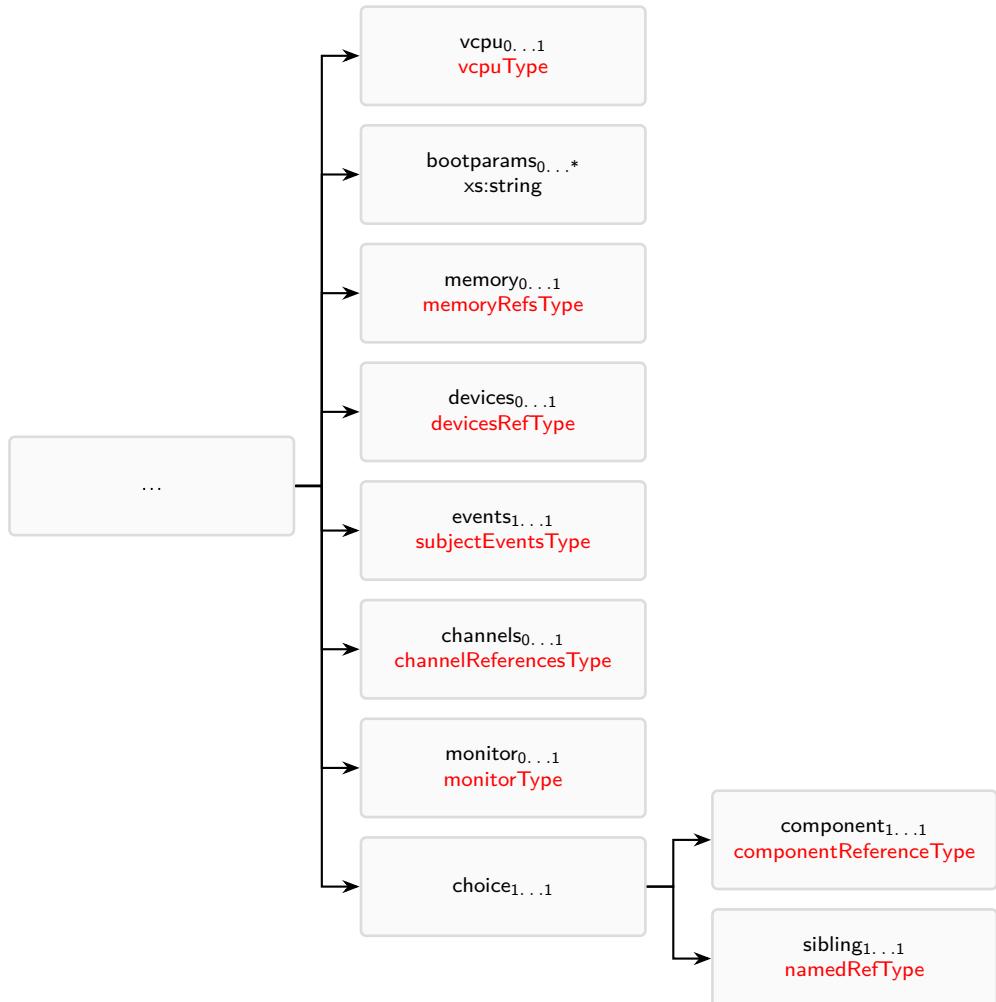
A subject is an instance of a component, i.e. an active component in the system policy that may be scheduled. Its specification references a component and maps all requested logical resources to physical resources provided by the system. Additional resources to the ones requested by the component can be specified here. This enables specialization of the base component specification.

See line 1852 in listing 8.1 for an example subject declaration.

Attributes

Name	Type	Use
name	nameType	required Unique subject name.

Structure



7.1.123 memoryRefsType

List of physical memory region references.

Structure



7.1.124 devicesRefType

List of device references. Used to grant a subject access to hardware devices and their resources.

See line 2191 in listing 8.1 for example device references.

Structure



7.1.125 deviceRefType

The device element allows a subject access to devices referenced via the physical attribute.

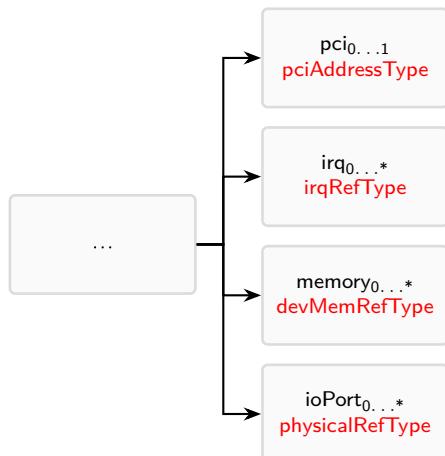
For PCI devices only a single virtual bus is provided (bus 0). The pci element may be used to place the device at a specific location (BDF). If no other logical device resources of the device are specified, then the expander tool will map all physical devices resources into the subject. When logical device resources are explicitly specified, then only access to those are actually granted. The physical attribute must be either a reference to an existing physical device, device alias or device class. Validators check that this is the case.

See line 2196 in listing 8.1 for an example reference.

Attributes

Name	Type	Use
logical	nameType	required Logical device name.
physical	nameType	required Name of physical device to reference.

Structure



7.1.126 pciAddressType

PCI Bus, Device, Function triplet (BDF).

Attributes

Name	Type	Use
bus	byteType	required PCI Bus number.
device	pciDeviceNumberType	required PCI Device number.
function	pciFunctionNumberType	required PCI Function number.

7.1.127 irqRefType

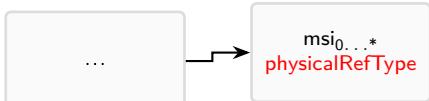
The device irq element assigns the referenced physical IRQ to the subject, i.e. if the device triggers the referenced physical IRQ, the specified vector number will be injected into the subject by the SK.

The presence of msi sub-elements enforces MSI mode (the default for MSI-capable devices and automatic device resource expansion).

Attributes

Name	Type	Use
logical	nameType	required Logical IRQ name.
physical	nameType	required Name of physical device IRQ.
vector	vectorType	optional Vector to inject into subject if device triggers IRQ. Will be allocated by the expander if none is specified.

Structure



7.1.128 physicalRefType

References a physical resource given by the physical attribute, and assigns a logical name to it.

Attributes

Name	Type	Use
logical	nameType	required Logical name for resource reference.
physical	nameType	required Name of physical resource.

7.1.129 devMemRefType

The device memory element maps the device memory region referenced via the physical attribute into the subject address space at address virtualAddress. The executable, writable attributes define the access permissions for the subject.

See line 2211 in listing 8.1 for an example device memory reference.

Attributes

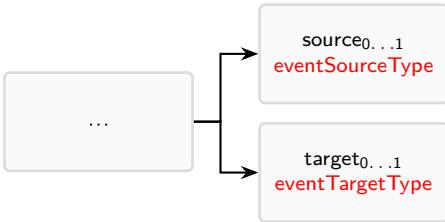
Name	Type	Use
virtualAddress	word64Type	optional Address of mapping in subject address space. If none is specified, an identity mapping is applied by the expander tool.
physical	nameType	required Name of referenced physical memory region.
logical	nameType	required Logical name of mapping.
writable	booleanType	required Defines if the mapped memory is writable.
executable	booleanType	required Defines if the memory region contents are executable by the processor.

7.1.130 subjectEventsType

The subject events element specifies all events originating from or directed at this subject. The physical attribute is the name of a event defined in the global events section.

See line 1861 in listing 8.1 for an example subject events section.

Structure



7.1.131 eventSourceType

The event source element specifies events that are allowed to be triggered by the associated subject.

Source events are divided into two groups: vmx_exit and vmcall. For event group vmx_exit the id attribute specifies the trap number while in the vmcall group it designates the hypercall number. For the valid range of IDs for each group see section [7.1.134](#).

The vmx_exit group is translated to a lookup table for handling VMX exit traps as defined by Intel SDM Vol. 3D, "Appendix C VMX Basic Exit Reasons". The vmcall group on the other hand is translated into a lookup table to handle hypercalls.

See line [1867](#) in listing [8.1](#) for an example event source section.

Structure



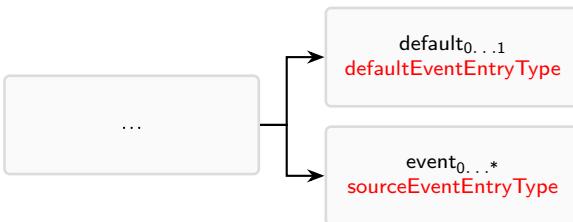
7.1.132 eventGroupType

Source event group element. Currently, two groups are supported: vmcall for hypercalls and vmx_exit for all other supported traps.

Attributes

Name	Type	Use
name	eventGroupNameType	required Name of event group.

Structure



7.1.133 defaultEventEntryType

Base: baseDefaultEventType

The default element entry can be used to specify an event which should be added for all event ids that have not been explicitly specified.

See line [1878](#) in listing [8.1](#) for a default source event example.

Attributes

Name	Type	Use
physical	nameType	required Global event reference.

(continuation)		
Name	Type	Use

7.1.134 sourceEventEntryType

Base: baseEventWithIDType < baseEventType

A source event entry specifies a source event node, i.e. it registers a handler for the given event id. These IDs, depending on the event group, are either hypercall numbers or VMX basic exit reasons. The valid ID ranges of the respective groups are:

vmx_exit 0 .. 59

vmcall 0 .. 63

Additionally, the following IDs in vmx_exit group are reserved and may not be used:

- Used by kernel: 1, 7, 41, 52, 55
- Reserved by Intel: 35, 38, 42

It is possible to assign event actions to event source entries. Currently supported source event actions are `subject_sleep`, `subject_yield`, `unmask_irq`, `system_reboot`, `system_poweroff` and `system_panic`, which all have the kernel itself as endpoint.

See line 1931 in listing 8.1 for a source event entry example.

Attributes

Name	Type	Use
logical	nameType	required
Logical event name.		
physical	nameType	required
Physical event name.		
id	xs:nonNegativeInteger	required
ID of event.		

7.1.135 eventGroupNameType

Base: xs:string

Supported event groups.

Restrictions

values:

- vmx_exit
- vmcall

7.1.136 eventTargetType

The event target element specifies events that the subject is an *endpoint* of.

See line 1961 in listing 8.1 for an example event target section.

Structure



7.1.137 targetEventEntryType

Base: baseEventType

The event element in the target section specifies one event endpoint by referencing a physical event and assigning a logical name to it.

See line 1966 in listing 8.1 for an example event endpoint.

Attributes

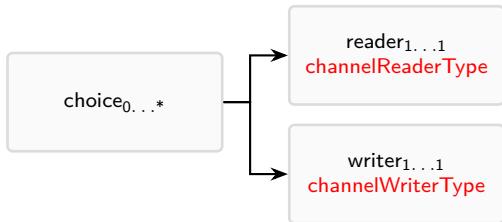
Name	Type	Use
logical	nameType	required
Logical event name.		
physical	nameType	required
Physical event name.		
id	xs:nonNegativeInteger	optional
Event ID.		

7.1.138 channelReferencesType

The channel section of a subject declares references to communication channels. The referenced channels become accessible to the requesting subject either as reader or writer endpoint.

See line 2325 in listing 8.1 for an example section.

Structure



7.1.139 channelReaderType

A channel reader element references a global communication channel as reader endpoint, i.e. the channel is mapped read-only into the subject address space.

See line 2331 in listing 8.1 for an example reader declaration.

Attributes

Name	Type	Use
logical	nameType	required
Logical name of reader channel.		
physical	nameType	required
Name of physical channel.		
virtualAddress	word64Type	required
Address of mapping in subject address space.		
vector	vectorType	optional
Associated vector. Must be set if a physical channel with hasEvent mode != switch is referenced (enforced by validator). The vector attribute is optional in the case of mode switch.		

7.1.140 channelWriterType

A channel writer element references a global communication channel as writer endpoint, i.e. the channel is mapped with write permissions into the subject address space.

See line 2338 in listing 8.1 for an example writer declaration.

Attributes

Name	Type	Use
logical	nameType	required Logical name of writer channel.
physical	nameType	required Name of physical channel.
virtualAddress	word64Type	required Address of mapping in subject address space.
event	xs:nonNegativeInteger	optional Associated event number. Must be set if a physical channel with hasEvent attribute is referenced.

7.1.141 monitorType

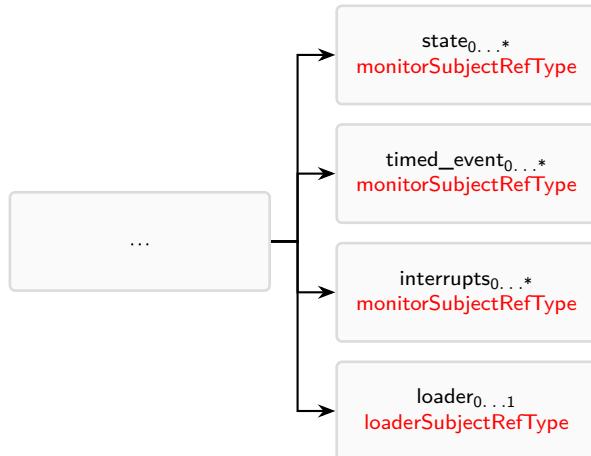
The monitor abstraction enables subjects to request access to certain data of another subject specified by name. Possible child elements are:

- State
- Timed_Events
- Interrupts
- Loader

See the Muen Component Specification document for details about these subject monitor interfaces.

See line 1977 in listing 8.1 for an example monitor section.

Structure



7.1.142 monitorSubjectRefType

Base: loaderSubjectRefType

Give subject monitor (SM) access to the referenced subject state.

Attributes

Name	Type	Use
subject	nameType	required Name of monitored subject.
logical	nameType	required Logical name of state mapping.
virtualAddress	word64Type	required Address to map requested subject address space.

(continuation)		
Name	Type	Use
writable	<code>booleanType</code>	required Whether or not the given state is mapped writable into the SM.

7.1.143 loaderSubjectRefType

The loader mechanism effectively puts the loaded subject denoted by the subject attribute under loader control, as it is not able to start without the help of the loader.

In more detail, the loader monitor element instructs the expander tool to map all memory regions of the referenced subject into the address space of the monitor subject, using the specified virtualAddress as offset in the address space of the loader.

If a memory region of the loaded subject is writable and file-backed, the region is replaced with an empty region and linked via the hashRef mechanism to the original region which is mapped into the loader.

The state of the loaded subject is then invalidated by clearing the CR4.VMXE bit in the initial subject CR4 register value. If such a subject is scheduled by the kernel, a VMX exit *VM-entry failure due to invalid guest state* (33) occurs. See Intel SDM Vol. 3C, "23.7 Enabling and Entering VMX Operation" and Intel SDM Vol. 3C, "23.8 Restrictions on VMX Operation" for more details. This trap is linked to the loader via normal VMX event handling. After handover, the loader initializes the memory regions replaced by the expander with the designated content.

All information required to *load* the loaded subject is provided to the loader subject via its own sinfo API. Memory regions prefixed with monitor_sinfo_ provide access to the sinfo regions of the loaded subjects. Regions prefixed with monitor_state_ specify memory regions containing the subject register state of the loaded subject.

The difference between the monitor_sinfo_ memory region address in the loader and the address of the sinfo memory region in the target sinfo information denotes the virtualAddress offset attribute of the loader element in the policy. This information combined is enough to fully construct the initial state of the loaded subject, or to reset a subject to its initial state on demand.

The loader may also optionally check the hashes of the restored regions, as this information is provided via the sinfo mechanism as well.

See line 1994 in listing 8.1 for an example loader element.

Attributes

Name	Type	Use
subject	<code>nameType</code>	required Name of monitored subject.
logical	<code>nameType</code>	required Logical name of state mapping.
virtualAddress	<code>word64Type</code>	required Address to map requested subject address space.

7.1.144 componentReferenceType

The component reference element specifies which component this subject instantiates. All logical resources required by the component must be mapped to physical resources of the appropriate type. Validators make sure that all requirements are satisfied and that no mapping has been omitted.

See line 1888 in listing 8.1 for an example component reference.

Attributes

Name	Type	Use
ref	<code>nameType</code>	required Name of referenced component.

Structure



7.1.145 resourceMappingType

The map element maps a physical resource provided by the system with a resource requested by the referenced component.

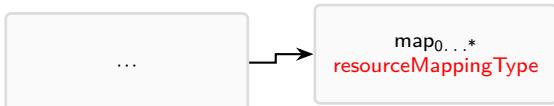
This element allows recursion to map child resources as well (e.g. device memory, I/O ports etc).

See line 1904 in listing 8.1 for an example mapping.

Attributes

Name	Type	Use
logical	nameType	required Name of logical resource requested by the component.
physical	nameType	required Physical name of resource.

Structure



7.1.146 schedulingType

The Muen SK implements a fixed, cyclic scheduler. The scheduling element is used to specify such a static plan by means of a major frame. A major frame consist of an arbitrary number of minor frames. Minor frames in turn specify a duration in number of ticks a scheduling partition is scheduled.

Scheduling partitions defined in the partitions element consist of one or more scheduling groups, which in turn specify one or more subjects to be scheduled. *Scheduling groups* are used to define groups of cooperating subjects, which are allowed to hand over execution to a subject in the same scheduling group. This is done via *handover* events. Membership of a scheduling group must be specified explicitly in the policy, validators enforce that these settings are correct by calculating the chain of handover events.

While scheduling groups support the efficient cooperation of multiple subjects, subjects which need to be spatially but not temporally isolated from each other cannot profit from it. To efficiently support this use-case, the scheduling partition concept is implemented.

Within a *scheduling partition*, all scheduling groups are scheduled round robin with preemption and the opportunity to yield and/or sleep. If a subject in a scheduling group sleeps or yields, the next scheduling group in the scheduling partition is scheduled. More precisely: the active subject of the next scheduling group is executed by the SK.

Note that prioritization is not implemented on purpose to avoid any starvation issues². The yield operation maps to the x86_64 PAUSE instruction, while sleep corresponds to HLT. See the *Muen Component Specification* document [1] for more information on this topic.

Minor frames designate the scheduling partition that is to be executed for the given amount of ticks. The scheduling partition attribute name uniquely identifies a scheduling partition. On first activation, the first scheduling group (in XML-order) is scheduled. Within the scheduling group, the first subject (again in XML-order) is executed. The active subject of a scheduling group may change over time, as the cooperating subjects initiate handover events.

The tickRate attribute of the scheduling element has the unit Hertz (Hz) and specifies the number of clock ticks per second. The ticks attribute of minor frames is expressed in terms of this

²Prioritization with starvation protection cannot be implemented with low complexity

tick rate. As an example: if we want to declare the minor frame duration in terms of microseconds (10^{-6}) then a tick rate of 1000000 must be used.

The duration of a major frame must be the same on each CPU, meaning the sum of all minor frame ticks for any given CPU must be identical. However, different major frames can have arbitrary length.

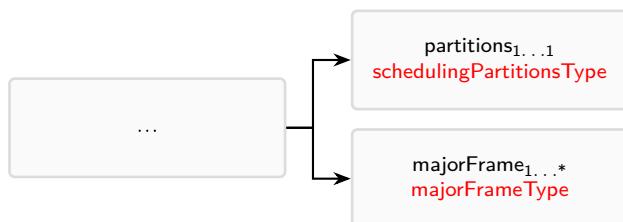
The Tau0 subject designates to the kernel which major frame is the currently active one. At the end of each major frame, the kernel determines the active major frame and switches to that scheduling plan for the duration of the major frame.

See line 2401 in listing 8.1 for an example scheduling plan.

Attributes

Name	Type	Use
tickRate	xs:positiveInteger	required Scheduling clock ticks in Hz.

Structure

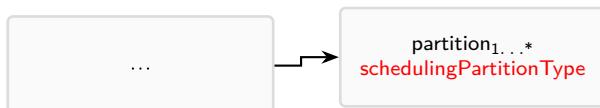


7.1.147 schedulingPartitionsType

The partitions element is used to specify all scheduling partitions of the system.

See line ?? in listing 8.1 for an example partitions element.

Structure



7.1.148 schedulingPartitionType

Base: baseSchedulingPartitionType

The scheduling partition element is used to specify a collection of scheduling groups consisting of subjects that require spatial but not temporal isolation from each other. Within a scheduling partition, all scheduling groups are scheduled round robin with preemption (i.e. non-cooperatively) and the opportunity to yield and/or sleep.

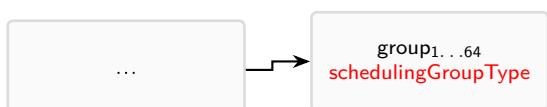
A scheduling partition must contain at least one scheduling group.

See line ?? in listing 8.1 for an example scheduling partition.

Attributes

Name	Type	Use
name	nameType	required Name of the scheduling partition.

Structure



7.1.149 schedulingGroupType

Base: baseSchedulingGroupType

The scheduling group element is used to specify a collection of subjects that may cooperatively schedule each other via handover events. Scheduling groups must contain at least one subject. As an example, a Linux subject and its associated Subject Monitor (SM), Subject Loader (SL) and Device Manager (DM) form a scheduling group.

See line ?? in listing 8.1 for an example scheduling group.

Structure



7.1.150 namedType

The namedType is used for simple elements in the policy, that only specify a name.

Attributes

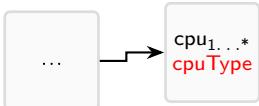
Name	Type	Use
name	nameType	required Name of element.

7.1.151 majorFrameType

A major frame consists of a sequence of minor frames for a given CPU. When the end of a major frame is reached, all CPUs synchronize and the scheduler starts over from the beginning using the first minor frame again. This means that major frames are repeated in a cyclic fashion until a different major frame is designated via the Tau0 interface.

See line 2520 in listing 8.1 for an example major frame.

Structure



7.1.152 cpuType

The cpu element is used to specify major frames for each CPU of the system.

See line 2528 in listing 8.1 for an example cpu element.

Attributes

Name	Type	Use
id	xs:nonNegativeInteger	required ID of CPU.

Structure



7.1.153 minorFrameType

A minor frame specifies the number of scheduling ticks a partition is allowed to run on the CPU specified by the parent cpu element.

See line 2533 in listing 8.1 for an example minor frame.

Attributes

Name	Type	Use
partition	nameType	required
Name of scheduled partition.		
ticks	xs:positiveInteger	required
Number of scheduling ticks in minor frame.		

Chapter 8

Appendix

8.1 Annotated Example Policy

```
1 <?xml version='1.0' encoding='utf-8'?>
2   <system>
3   <!--
4     A Muen system policy specifies all hardware resources such as physical
5       memory, devices, CPU time, etc and how these resources are accessed by
6       the separation kernel, the subjects and devices.
7
8     The 'system' section is the top-level element in the Muen system policy.
9     It contains various sub-elements which specify all aspects of a concrete
10    system.
11
12    This is the *source format* of the Muen system policy. It allows for
13    abstractions, such as channels, which are broken down into their
14    constituent parts by the toolchain in format A and B accordingly.
15    -->
16    <config>
17    <!--
18      The purpose of a config section is to specify configuration values which
19      parameterize a system or a component. It allows to declare boolean,
20      string and integer values. The following sections in the system policy
21      provide support for configuration values:
22
23    - System
24
25    - Platform
26
27    - Component
28
29    During the build process, configuration values provided by the platform
30    are merged into the global system configuration. Component configuration
31    values allow the parameterization of component-local functionality.
32
33    Besides component parameterization, configuration options can be used in
34    'if' conditionals, as shown in the following example.
35
36    ```` xml
37    <if variable="xhcidbg_enabled" value="true">
38    ...
39    </if>
40    ````
41
42    If the type of the referenced variable is string the comparison is
43    case-sensitive. A second use case is XML attribute value expansion as
44    follows:
45
46    ```` xml
47    <channel name="debuglog" size="$logchannel_size"/>
48    ````
49
50    The 'size' attribute value is not specified directly, but parameterized
51    via an integer configuration option.
52    -->
53    <boolean name="pciconf_emulation_enabled" value="true"/>
54    <boolean name="pciconf_emulation_xhci_enabled" value="false"/>
55    <boolean name="xhcidbg_supported" value="false"/>
56    <boolean name="xhcidbg_enabled" value="true"/>
57    <boolean name="dbgserver_sink_serial" value="true"/>
58    <boolean name="ahci_supported" value="true"/>
59    <boolean name="serial_supported" value="true"/>
60    <boolean name="dbgserver_sink_xhcidbg" value="false"/>
61    <boolean name="linux_debug" value="false"/>
```

```

63   <boolean name="ahci_drv_enabled" value="false"/>
64   <boolean name="dbgserver_sink_shmem" value="false"/>
65   <boolean name="uefi_gop_rmrr_access" value="false"/>
66   <boolean name="huart_supported" value="false"/>
67   <boolean name="ahci_drv_active" value="false"/>
68   <boolean name="dbgserver_sink_pcspkr" value="false"/>
69   <boolean name="dbgserver_serial_enabled" value="true"/>
70   <string name="pciconf_emulation_xhci_devid" value="16#02#/"/>
71   <string name="pciconf_emulation_nic_devid" value="16#01#/"/>
72   <string name="logchannel_size" value="16#0002_0000#/"/>
73   <string name="platform" value="platform/lenovo-t430s.xml"/>
74   <string name="hardware" value="hardware/lenovo-t430s.xml"/>
75   <string name="pciconf_emulation_xhci_physdev" value="usb_controller_1"/>
76   <string name="system" value="xml/demo_system_vtd.xml"/>
77   <string name="pciconf_emulation_nic_physdev" value="ethernet_controller_1"/>
78   <string name="igd_opregion_address" value="16#baf5_5000#/"/>
79   <string name="additional_hardware" value="hardware/common_hardware.xml"/>
80   </config>
81   </hardware>
82   <!--
83     Systems running the Muen SK perform static resource allocation at
84     integration time. This means that all available hardware resources of a
85     target machine must be defined in the system policy in order for these
86     resources to be allocated to subjects.
87
88     The 'hardware' element is the top-level element of the hardware
89     specification in the system policy. Information provided by a hardware
90     description includes the amount of available physical memory blocks
91     including reserved memory regions (RMRR), the number of physical CPU
92     cores and hardware device resources.
93
94     The Muen toolchain provides a handy tool to automate the cumbersome
95     process of gathering hardware resource data from a running Linux system:
96     'mugenhwcfg`[1\].
97
98     1. https://git.codelabs.ch/?p=muen/mugenhwcfg.git
99     -->
100    <processor cpuCores="2" speed="2893431" vmxTimerRate="5">
101    <!--
102      The 'processor' element specifies the number of CPU cores, the processor
103      speed in kHz and the Intel VMX preemption timer rate.
104
105      Since Intel CPUs can have arbitrary APIC identifiers, the APIC IDs of
106      all physical CPUs are enumerated here. The APIC ID is required for
107      interrupt and IPI routing.
108
109      The 'processor' element also lists register values for all CPUID leaves
110      of the hardware target, and some MSR values of interest.
111      -->
112      <cpu apicId="0"/>
113      <cpu apicId="2"/>
114      <cpuid eax="16#0000_000d#" ebx="16#756e_6547#" ecx="16#6c65_746e#" edx="16#4965_6e69#" leaf="16#0000
115          _0000#" subleaf="16#00#"/>
116      <cpuid eax="16#0003_06a9#" ebx="16#0010_0800#" ecx="16#7fba_e3ff#" edx="16#bfbe_fbff#" leaf="16#0000
117          _0001#" subleaf="16#00#"/>
118      <cpuid eax="16#7603_5a01#" ebx="16#00f0_b2ff#" ecx="16#0000_0000#" edx="16#00ca_0000#" leaf="16#0000
119          _0002#" subleaf="16#00#"/>
120      <cpuid eax="16#0000_0000#" ebx="16#0000_0000#" ecx="16#0000_0000#" edx="16#0000_0000#" leaf="16#0000
121          _0003#" subleaf="16#00#"/>
122      <cpuid eax="16#1c00_4121#" ebx="16#01c0_003f#" ecx="16#0000_003f#" edx="16#0000_0000#" leaf="16#0000
123          _0004#" subleaf="16#00#"/>
124      <cpuid eax="16#1c00_4122#" ebx="16#01c0_003f#" ecx="16#0000_003f#" edx="16#0000_0000#" leaf="16#0000
125          _0004#" subleaf="16#01#"/>
126      <cpuid eax="16#1c00_4143#" ebx="16#01c0_003f#" ecx="16#0000_01ff#" edx="16#0000_0000#" leaf="16#0000
127          _0004#" subleaf="16#02#"/>
128      <cpuid eax="16#1c03_c163#" ebx="16#03c0_003f#" ecx="16#0000_0fff#" edx="16#0000_0006#" leaf="16#0000
129          _0004#" subleaf="16#03#"/>
130      <cpuid eax="16#0000_0040#" ebx="16#0000_0040#" ecx="16#0000_0003#" edx="16#0002_1120#" leaf="16#0000
131          _0005#" subleaf="16#00#"/>
132      <cpuid eax="16#0000_0077#" ebx="16#0000_0002#" ecx="16#0000_0009#" edx="16#0000_0000#" leaf="16#0000
133          _0006#" subleaf="16#00#"/>
134      <cpuid eax="16#0000_0000#" ebx="16#0000_0281#" ecx="16#0000_0000#" edx="16#9c00_0400#" leaf="16#0000
135          _0007#" subleaf="16#00#"/>
136      <cpuid eax="16#0000_0000#" ebx="16#0000_0000#" ecx="16#0000_0000#" edx="16#0000_0000#" leaf="16#0000
137          _0008#" subleaf="16#00#"/>
138      <cpuid eax="16#0000_0000#" ebx="16#0000_0000#" ecx="16#0000_0000#" edx="16#0000_0000#" leaf="16#0000
139          _0009#" subleaf="16#00#"/>
140      <cpuid eax="16#0730_0403#" ebx="16#0000_0000#" ecx="16#0000_0000#" edx="16#0000_0603#" leaf="16#0000
141          _000a#" subleaf="16#00#"/>
142      <cpuid eax="16#0000_0001#" ebx="16#0000_0002#" ecx="16#0000_0100#" edx="16#0000_0000#" leaf="16#0000
143          _000b#" subleaf="16#00#"/>
144      <cpuid eax="16#0000_0004#" ebx="16#0000_0004#" ecx="16#0000_0201#" edx="16#0000_0000#" leaf="16#0000
145          _000b#" subleaf="16#01#"/>
146      <cpuid eax="16#0000_0000#" ebx="16#0000_0000#" ecx="16#0000_0000#" edx="16#0000_0000#" leaf="16#0000
147          _000c#" subleaf="16#00#"/>
148      <cpuid eax="16#0000_0007#" ebx="16#0000_0340#" ecx="16#0000_0340#" edx="16#0000_0000#" leaf="16#0000
149          _000d#" subleaf="16#00#"/>
```

```

131  <cpuid eax="16#0000_0001#" ebx="16#0000_0000#" ecx="16#0000_0000#" edx="16#0000_0000#" leaf="16#0000_000d#" subleaf="16#01#" />
132  <cpuid eax="16#0000_0100#" ebx="16#0000_0240#" ecx="16#0000_0000#" edx="16#0000_0000#" leaf="16#0000_000d#" subleaf="16#02#" />
133  <cpuid eax="16#0000_0007#" ebx="16#0000_0340#" ecx="16#0000_0340#" edx="16#0000_0000#" leaf="16#2000_0000#" subleaf="16#00#" />
134  <cpuid eax="16#8000_0008#" ebx="16#0000_0000#" ecx="16#0000_0000#" edx="16#0000_0000#" leaf="16#8000_0000#" subleaf="16#00#" />
135  <cpuid eax="16#0000_0000#" ebx="16#0000_0000#" ecx="16#0000_0001#" edx="16#2810_0800#" leaf="16#8000_0001#" subleaf="16#00#" />
136  <cpuid eax="16#2020_2020#" ebx="16#4920_2020#" ecx="16#6c65_746e#" edx="16#2029_5228#" leaf="16#8000_0002#" subleaf="16#00#" />
137  <cpuid eax="16#6572_6f43#" ebx="16#294d_5428#" ecx="16#2d37_6920#" edx="16#3032_3533#" leaf="16#8000_0003#" subleaf="16#00#" />
138  <cpuid eax="16#5043_204d#" ebx="16#2040_2055#" ecx="16#3039_2e32#" edx="16#007a_4847#" leaf="16#8000_0004#" subleaf="16#00#" />
139  <cpuid eax="16#0000_0000#" ebx="16#0000_0000#" ecx="16#0000_0000#" edx="16#0000_0000#" leaf="16#8000_0005#" subleaf="16#00#" />
140  <cpuid eax="16#0000_0000#" ebx="16#0000_0000#" ecx="16#0100_6040#" edx="16#0000_0000#" leaf="16#8000_0006#" subleaf="16#00#" />
141  <cpuid eax="16#0000_0000#" ebx="16#0000_0000#" ecx="16#0000_0000#" edx="16#0000_0100#" leaf="16#8000_0007#" subleaf="16#00#" />
142  <cpuid eax="16#0000_3024#" ebx="16#0000_0000#" ecx="16#0000_0000#" edx="16#0000_0000#" leaf="16#8000_0008#" subleaf="16#00#" />
143  <cpuid eax="16#0000_0007#" ebx="16#0000_0340#" ecx="16#0000_0340#" edx="16#0000_0000#" leaf="16#8086_0000#" subleaf="16#00#" />
144  <cpuid eax="16#0000_0007#" ebx="16#0000_0340#" ecx="16#0000_0340#" edx="16#0000_0000#" leaf="16#0000_0000#" subleaf="16#00#" />
145  <msr address="16#0000_003a#" name="IA32_FEATURE_CONTROL" regval="16#0000_0000_0000_0005#" />
146  <msr address="16#0000_01a0#" name="IA32_MISC_ENABLE" regval="16#0000_0000_0085_0089#" />
147  <msr address="16#0000_0480#" name="IA32_VMX_BASIC" regval="16#00da_0400_0000_0010#" />
148  <msr address="16#0000_0481#" name="IA32_VMX_PINBASED_CTLs" regval="16#0000_007f_0000_0016#" />
149  <msr address="16#0000_0482#" name="IA32_VMX_PROCBASED_CTLs" regval="16#ffff9_ffff_e041_e172#" />
150  <msr address="16#0000_0483#" name="IA32_VMX_EXIT_CTLs" regval="16#007f_ffff_0003_6dff#" />
151  <msr address="16#0000_0484#" name="IA32_VMX_ENTRY_CTLs" regval="16#0000_ffff_0000_11ff#" />
152  <msr address="16#0000_0485#" name="IA32_VMX_MISC" regval="16#0000_0000_1004_01e5#" />
153  <msr address="16#0000_0486#" name="IA32_VMX_CRO_FIXED0" regval="16#0000_0000_8000_0021#" />
154  <msr address="16#0000_0487#" name="IA32_VMX_CRO_FIXED1" regval="16#0000_0000_ffff_ffff#" />
155  <msr address="16#0000_0488#" name="IA32_VMX_CR4_FIXED0" regval="16#0000_0000_0000_2000#" />
156  <msr address="16#0000_0489#" name="IA32_VMX_CR4_FIXED1" regval="16#0000_0000_0017_67ff#" />
157  <msr address="16#0000_048b#" name="IA32_VMX_PROCBASED_CTLs2" regval="16#0000_08ff_0000_0000#" />
158  <msr address="16#0000_048c#" name="IA32_VMX_EPT_VPID_CAP" regval="16#0000_0f01_0611_4141#" />
159  <msr address="16#0000_048d#" name="IA32_VMX_TRUE_PINBASED_CTLs" regval="16#0000_007f_0000_0016#" />
160  <msr address="16#0000_048e#" name="IA32_VMX_TRUE_PROCBASED_CTLs" regval="16#ffff9_ffff_e040_6172#" />
161  <msr address="16#0000_048f#" name="IA32_VMX_TRUE_EXIT_CTLs" regval="16#007f_ffff_0003_6dfb#" />
162  <msr address="16#0000_0490#" name="IA32_VMX_TRUE_ENTRY_CTLs" regval="16#0000_ffff_0000_11fb#" />
163  </processor>
164  <memory>
165  <!--
166   The hardware 'memory' element specifies the available physical memory
167   blocks including reserved memory regions (RMRR, see Intel VT-d
168   Specification, "8.4 Reserved Memory Region Reporting Structure").
169
170   Only memory blocks reported by the BIOS E820 map as non-*reserved* must
171   be configured in this section, e.g. *usable* or *ACPI NVS*, *ACPI data*.
172   -->
173  <memoryBlock allocatable="false" name="System RAM" physicalAddress="16#0000#" size="16#0009_d000#" />
174  <memoryBlock allocatable="true" name="System RAM" physicalAddress="16#0010_0000#" size="16#1ff0_0000#" />
175  <memoryBlock allocatable="true" name="System RAM" physicalAddress="16#2020_0000#" size="16#1fe0_4000#" />
176  <memoryBlock allocatable="true" name="System RAM" physicalAddress="16#4000_5000#" size="16#6ed2_c000#" />
177  <memoryBlock allocatable="false" name="ACPI Non-volatile Storage" physicalAddress="16#bae9_f000#" size="16#0010_0000#" />
178  <memoryBlock allocatable="false" name="ACPI Tables" physicalAddress="16#baf9_f000#" size="16#0006_0000#" />
179  <memoryBlock allocatable="true" name="System RAM" physicalAddress="16#0001_0000_0000#" size="16#0003_3e60_0000#" />
180  <reservedMemory name="rmrr1" physicalAddress="16#ba3b_a000#" size="16#0001_7000#" />
181  <!--
182   A 'reservedMemory' element is a special memory block declaration. It
183   specifies a reserved memory region as outlined in the Intel VT-d
184   Specification, "8.4 Reserved Memory Region Reporting Structure" (RMRR).
185
186   Reserved memory regions are BIOS allocated memory ranges that may be DMA
187   targets for certain legacy device use-cases. Devices that require access
188   to such a region refer to it by name.
189   -->
190  </reservedMemory>
191  <reservedMemory name="rmrr2" physicalAddress="16#bb80_0000#" size="16#0420_0000#" />
192  </memory>
193  <devices pciConfigAddress="16#f800_0000#" pciConfigSize="16#0400_0000#" />
194  <!--
195   The 'devices' element enumerates all devices provided by the hardware
196   platform. Different kinds of devices, be it PCI(e) or legacy (non-PCI),
197   can be declared in this section.

```

```

-->
199 <device name="vga">
<!--
201 The 'device' element specifies a physical device and its associated
resources. There are three main device resource types:
203 - IRQ
205 - I/O port range
207 - Memory
209
The presence of a PCI element indicates whether the device is a PCI or a
211 legacy device.
213 Capabilities can be used to convey additional device-specific
information. The base address of the memory mapped PCI config space is
215 defined by the 'pciConfigAddress' attribute.
-->
217 <memory caching="WC" name="buffer" physicalAddress="16#000a_0000#" size="16#0002_0000#">
<!--
219 A device 'memory' element specifies a memory region which is used to
interact with the associated device.
221 For PCI devices, the specified region is programmed into one device BAR
223 (Base Address Register) by system firmware. See the PCI Local Bus
Specification or the PCI Express Base Specification for more details.
225 -->
</memory>
227 <ioPort end="16#03df#" name="ports" start="16#03c0#">
<!--
229 The 'ioPort' element specifies a device I/O port resource from 'start'
octet up to and including 'end' octet. A single byte-accessed port is
231 designated by specifying the same 'start' and 'end' values.
-->
233 </ioPort>
</device>
235 <device name="ps2">
<irq name="kbd_irq" number="1">
237 <!--
The 'irq' element specifies a device IRQ resource.
239
The specified IRQ number is one of:
241 - Legacy IRQ (ISA)
Range '0 .. 15'.
245 - PCI INTx IRQ, line-signaled
Range '0 .. Max_LSI_IRQ', whereas 'Max_LSI_IRQ' is defined by the
247 hardware I/O APIC configuration 'gsi_base' + 'max_redirection_entry'
of I/O APIC with 'max(gsi_base)'. 'gsi_base' and
'max_redirection_entry' are I/O APIC device capabilities.
251 'msi' sub-elements are present if the device supports MSI interrupts.
The element count designates the number of supported MSI interrupts.
-->
253 </irq>
255 <irq name="mouse_irq" number="12"/>
<ioPort end="16#0060#" name="port_60" start="16#0060#">
257 <ioPort end="16#0064#" name="port_64" start="16#0064#">
</device>
259 <device name="cmos_rtc">
<ioPort end="16#0071#" name="ports" start="16#0070#">
261 </device>
<device name="pcspeaker">
263 <ioPort end="16#0061#" name="Port_61" start="16#0061#">
<ioPort end="16#0043#" name="Port_42_43" start="16#0042#">
265 </device>
<device name="system_board">
267 <!--
The system board must provide a reset and pmla_cnt port as well as
269 the pmla_cnt_slp_typ capability. The presence of this device and
the necessary resources are checked by the Mucfgvalidate tool. The
271 resources are used by the kernel for system reboot and poweroff.
-->
273 <ioPort end="16#0cf9#" name="reset" start="16#0cf9#">
<ioPort end="16#0404#" name="pmla_cnt" start="16#0404#">
275 <capabilities>
<capability name="systemboard"/>
277 <capability name="pmla_cnt_slp_typ">7168</capability>
</capabilities>
279 </device>
<device name="ioapic_1">
281 <!--
The I/O Advanced Programmable Interrupt Controller (I/O APIC) is
283 used by the kernel for interrupt routing of legacy IRQs. The
presence of this device and the necessary resources are checked by

```

```

285 the validator tool.
-->
287 <memory caching="UC" name="mem1" physicalAddress="16#fec0_0000#" size="16#1000#" />
288 <capabilities>
289 <capability name="ioapic">
<!--
291 A device 'capability' is used to assign additional information to a
293 device. Such a capability might be used by the Muen toolchain to perform
295 certain actions on devices with a given capability (e.g. 'ioapic'). A
296 system integrator may use this facility to define its own capabilities
297 used by custom tools.
298
299 A capability element can have an optional value.
-->
300 </capability>
301 <capability name="gsi_base">0</capability>
302 <capability name="max_redirection_entry">23</capability>
303 <capability name="source_id">16#f0f8#</capability>
304 </capabilities>
305 </device>
306 <device name="iommu_1">
<!--
307 This device specifies an Intel VT-d DMA and interrupt remapping
308 hardware. It is used by the Muen SK to implement device separation
309 by means of device domains, see below. The capabilities define
310 specific properties of the IOMMU, such as Guest Address Width,
311 Fault Register Offset etc. Refer to the Intel VT-d Specification,
312 "10.4 Register Descriptions".
-->
313 <memory caching="UC" name="mmio" physicalAddress="16#fed9_0000#" size="16#1000#" />
314 <capabilities>
315 <capability name="iommu"/>
316 <capability name="agaw">39</capability>
317 <capability name="fr_offset">512</capability>
318 <capability name="iotlb_invalidate_offset">264</capability>
319 </capabilities>
320 </device>
321 <device name="iommu_2">
322 <memory caching="UC" name="mmio" physicalAddress="16#fed9_1000#" size="16#1000#" />
323 <capabilities>
324 <capability name="iommu"/>
325 <capability name="agaw">39</capability>
326 <capability name="fr_offset">512</capability>
327 <capability name="iotlb_invalidate_offset">264</capability>
328 </capabilities>
329 </device>
330 <device name="host_bridge_1">
331 <description>Intel Corporation 3rd Gen Core processor DRAM Controller</description>
332 <pci bus="16#00#" device="16#00#" function="0">
<!--
333 PCI(e) devices are specified using the 'pci' element.
334
335 The element provides the following information:
336
337 - PCI device address (BDF)
338 - Identification
339 - IOMMU group information
340
341 The location of the PCI device in the PCI topology is specified by the
342 Bus, Device, Function triplet (BDF).
-->
343 <identification classcode="16#0600#" deviceId="16#0154#" revisionId="16#09#" vendorId="16#8086#" />
344 <!--
345 The 'identification' element specifies the PCI device class, device,
346 revision and vendor ID.
347
348 For more information, consult the PCI Local Bus Specification,
349 "Configuration Space Decoding".
-->
350 </identification>
351 <iommuGroup id="0">
<!--
352 Devices in the same IOMMU group cannot be properly isolated from each
353 other because they may perform inter-device transactions directly,
354 without going through the IOMMU.
355
356 Note that this information is currently not used by the toolchain. It is
357 a hint to the system integrator whether two devices can be properly
358 isolated from each other or not.
-->
359 </iommuGroup>
360 </pci>
361 <memory caching="UC" name="mmconf" physicalAddress="16#f800_0000#" size="16#1000#" />
362 </device>
363 <device name="vga_compatible_controller_1">

```

```

<description>Intel Corporation 3rd Gen Core processor Graphics Controller</description>
373 <pci bus="16#00#" device="16#02#" function="0">
374 <identification classcode="16#0300#" deviceId="16#0166#" revisionId="16#09#" vendorId="16#8086#/>
375 <iommuGroup id="1"/>
376 </pci>
377 <irq name="irq1" number="16">
378 <msi name="msil"/>
379 </irq>
380 <memory caching="UC" name="mem1" physicalAddress="16#d000_0000#" size="16#0040_0000#"/>
381 <memory caching="WC" name="mem2" physicalAddress="16#c000_0000#" size="16#1000_0000#"/>
382 <memory caching="WC" name="mem3" physicalAddress="16#00c_0000#" size="16#0002_0000#"/>
383 <memory caching="UC" name="mnconf" physicalAddress="16#f801_0000#" size="16#1000#"/>
384 <ioPort end="16#603f#" name="ioport1" start="16#6000#"/>
385 <reservedMemory ref="rmrr2"/>
386 <!--
387 This device specifies that it requires access to the reserved
388 memory range (RMRR) with the given name.
389 -->
390 </reservedMemory>
391 </device>
392 <device name="usb_controller_1">
393 <description>Intel Corporation 7 Series/C210 Series Chipset Family USB xHCI Host Controller</
394 description>
395 <pci bus="16#00#" device="16#14#" function="0">
396 <identification classcode="16#0c03#" deviceId="16#1e31#" revisionId="16#04#" vendorId="16#8086#/>
397 <iommuGroup id="2"/>
398 </pci>
399 <irq name="irq1" number="16">
400 <msi name="msil"/>
401 <!--
402 There are two different interrupt types which devices may trigger:
403 legacy/PCI LSI IRQs and Message Signaled Interrupts (MSI). The
404 legacy/PCI LSI IRQ is specified by the number attribute of the 'irq'
405 element. For MSIs, each 'msi' element defines an MSI IRQ that may be
406 assigned to subjects. Each MSI may be individually routed.
407 -->
408 </msi>
409 <msi name="msi2"/>
410 <msi name="msi3"/>
411 <msi name="msi4"/>
412 <msi name="msi5"/>
413 <msi name="msi6"/>
414 <msi name="msi7"/>
415 <msi name="msi8"/>
416 </irq>
417 <memory caching="UC" name="mem1" physicalAddress="16#d252_0000#" size="16#0001_0000#"/>
418 <memory caching="UC" name="mnconf" physicalAddress="16#f80a_0000#" size="16#1000#"/>
419 <reservedMemory ref="rmrr1"/>
420 </device>
421 <device name="communication_controller_1">
422 <description>Intel Corporation 7 Series/C216 Chipset Family MEI Controller #1</description>
423 <pci bus="16#00#" device="16#16#" function="0">
424 <identification classcode="16#0780#" deviceId="16#1e3a#" revisionId="16#04#" vendorId="16#8086#/>
425 <iommuGroup id="3"/>
426 </pci>
427 <irq name="irq1" number="16">
428 <msi name="msil"/>
429 </irq>
430 <memory caching="UC" name="mem1" physicalAddress="16#d253_5000#" size="16#1000#"/>
431 <memory caching="UC" name="mnconf" physicalAddress="16#f80b_0000#" size="16#1000#"/>
432 </device>
433 <device name="serial_controller_1">
434 <description>Intel Corporation 7 Series/C210 Series Chipset Family KT Controller</description>
435 <pci bus="16#00#" device="16#16#" function="3">
436 <identification classcode="16#0700#" deviceId="16#1e3d#" revisionId="16#04#" vendorId="16#8086#/>
437 <iommuGroup id="3"/>
438 </pci>
439 <irq name="irq1" number="19">
440 <msi name="msil"/>
441 </irq>
442 <memory caching="UC" name="mem1" physicalAddress="16#d253_c000#" size="16#1000#"/>
443 <memory caching="UC" name="mnconf" physicalAddress="16#f80b_3000#" size="16#1000#"/>
444 <ioPort end="16#60b7#" name="ioport1" start="16#60b0#"/>
445 </device>
446 <device name="ethernet_controller_1">
447 <description>Intel Corporation 82579LM Gigabit Network Connection (Lewisville)</description>
448 <pci bus="16#00#" device="16#19#" function="0">
449 <identification classcode="16#0200#" deviceId="16#1502#" revisionId="16#04#" vendorId="16#8086#/>
450 <iommuGroup id="4"/>
451 </pci>
452 <irq name="irq1" number="20">
453 <msi name="msil"/>
454 </irq>
455 <memory caching="UC" name="mem1" physicalAddress="16#d250_0000#" size="16#0002_0000#"/>
456 <memory caching="UC" name="mem2" physicalAddress="16#d253_b000#" size="16#1000#"/>
457 <memory caching="UC" name="mnconf" physicalAddress="16#f80c_8000#" size="16#1000#"/>
458 <ioPort end="16#609f#" name="ioport1" start="16#6080#"/>

```

```

    </device>
549 <device name="usb_controller_2">
550   <description>Intel Corporation 7 Series/C216 Chipset Family USB Enhanced Host Controller #2</
      description>
551   <pci bus="16#00#" device="16#1a#" function="0">
552     <identification classcode="16#0c03#" deviceId="16#1e2d#" revisionId="16#04#" vendorId="16#8086#" />
553     <iommuGroup id="5"/>
554   </pci>
555   <irq name="irq1" number="16"/>
556   <memory caching="UC" name="mem1" physicalAddress="16#d253_a000#" size="16#1000#" />
557   <memory caching="UC" name="mmconf" physicalAddress="16#f80d_0000#" size="16#1000#" />
558   <reservedMemory ref="rmrr1"/>
559 </device>
560 <device name="audio_device_1">
561   <description>Intel Corporation 7 Series/C216 Chipset Family High Definition Audio Controller</
      description>
562   <pci bus="16#00#" device="16#1b#" function="0">
563     <identification classcode="16#0403#" deviceId="16#1e20#" revisionId="16#04#" vendorId="16#8086#" />
564     <iommuGroup id="6"/>
565   </pci>
566   <irq name="irq1" number="22">
567     <msi name="msil"/>
568   </irq>
569   <memory caching="UC" name="mem1" physicalAddress="16#d253_0000#" size="16#4000#" />
570   <memory caching="UC" name="mmconf" physicalAddress="16#f80d_8000#" size="16#1000#" />
571 </device>
572 <device name="usb_controller_3">
573   <description>Intel Corporation 7 Series/C216 Chipset Family USB Enhanced Host Controller #1</
      description>
574   <pci bus="16#00#" device="16#1d#" function="0">
575     <identification classcode="16#0c03#" deviceId="16#1e26#" revisionId="16#04#" vendorId="16#8086#" />
576     <iommuGroup id="11"/>
577   </pci>
578   <irq name="irq1" number="23"/>
579   <memory caching="UC" name="mem1" physicalAddress="16#d253_9000#" size="16#1000#" />
580   <memory caching="UC" name="mmconf" physicalAddress="16#f80e_8000#" size="16#1000#" />
581   <reservedMemory ref="rmrr1"/>
582 </device>
583 <device name="isa_bridge_1">
584   <description>Intel Corporation QM77 Express Chipset LPC Controller</description>
585   <pci bus="16#00#" device="16#1f#" function="0">
586     <identification classcode="16#0601#" deviceId="16#1e55#" revisionId="16#04#" vendorId="16#8086#" />
587     <iommuGroup id="12"/>
588   </pci>
589   <memory caching="UC" name="mmconf" physicalAddress="16#f80f_8000#" size="16#1000#" />
590 </device>
591 <device name="sata_controller_1">
592   <description>Intel Corporation 7 Series Chipset Family 6-port SATA Controller [AHCI mode]</
      description>
593   <pci bus="16#00#" device="16#1f#" function="2">
594     <identification classcode="16#0106#" deviceId="16#1e03#" revisionId="16#04#" vendorId="16#8086#" />
595     <iommuGroup id="12"/>
596   </pci>
597   <irq name="irq1" number="19">
598     <msi name="msil"/>
599   </irq>
600   <memory caching="UC" name="mem1" physicalAddress="16#d253_8000#" size="16#1000#" />
601   <memory caching="UC" name="mmconf" physicalAddress="16#f80f_a000#" size="16#1000#" />
602   <ioPort end="16#60af#" name="ioport1" start="16#60a8#"/>
603   <ioPort end="16#60bf#" name="ioport2" start="16#60bc#"/>
604   <ioPort end="16#60a7#" name="ioport3" start="16#60a0#"/>
605   <ioPort end="16#60bb#" name="ioport4" start="16#60b8#"/>
606   <ioPort end="16#607f#" name="ioport5" start="16#6060#"/>
607 </device>
608 <device name="smbus_1">
609   <description>Intel Corporation 7 Series/C216 Chipset Family SMBus Controller</description>
610   <pci bus="16#00#" device="16#1f#" function="3">
611     <identification classcode="16#0c05#" deviceId="16#1e22#" revisionId="16#04#" vendorId="16#8086#" />
612     <iommuGroup id="12"/>
613   </pci>
614   <irq name="irq1" number="18"/>
615   <memory caching="UC" name="mem1" physicalAddress="16#d253_4000#" size="16#1000#" />
616   <memory caching="UC" name="mmconf" physicalAddress="16#f80f_b000#" size="16#1000#" />
617   <ioPort end="16#efbf#" name="ioport1" start="16#efa0#"/>
618 </device>
619 <device name="network_controller_1">
620   <description>Intel Corporation Centrino Advanced-N 6205 [Taylor Peak]</description>
621   <pci bus="16#03#" device="16#00#" function="0">
622     <identification classcode="16#0280#" deviceId="16#0085#" revisionId="16#34#" vendorId="16#8086#" />
623     <iommuGroup id="13"/>
624   </pci>
625   <irq name="irq1" number="17">
626     <msi name="msil"/>
627   </irq>
628   <memory caching="UC" name="mem1" physicalAddress="16#dlc0_0000#" size="16#2000#" />
629   <memory caching="UC" name="mmconf" physicalAddress="16#f830_0000#" size="16#1000#" />
630 </device>

```

```

541 <device name="system_peripheral_1">
542   <description>Ricoh Co Ltd PCIe SDXC/MMC Host Controller</description>
543   <pci bus="16#04#" device="16#00#" function="0">
544     <identification classcode="16#0880#" deviceId="16#e823#" revisionId="16#07#" vendorId="16#1180#" />
545   <iommuGroup id="14"/>
546 </pci>
547 <irq name="irq1" number="18">
548   <msti name="msil"/>
549 </irq>
550   <memory caching="UC" name="mem1" physicalAddress="16#d140_0000#" size="16#1000#"/>
551   <memory caching="UC" name="mmconf" physicalAddress="16#f840_0000#" size="16#1000#"/>
552 </device>
553 </devices>
554 </hardware>
555 <platform>
556 <!--
557 To enable an uniform view of the hardware resources across different
558 physical machines from the system integrators perspective, the platform
559 description layer is interposed between the hardware resource
560 description and the rest of the system policy. This allows to build a
561 Muen system for different physical target machines using the same system
562 policy.
563 -->
564 <mappings>
565 <!--
566 Platform device alias and class mappings section. Used to assign a
567 stable name to a hardware device or to group (multiple) devices under a
568 given name.
569 -->
570 <aliases>
571 <!--
572 Aliases are a renaming mechanism for physical hardware devices and their
573 resources. By using alias names in the system policy references to
574 concrete hardware resources can be avoided. Additionally, aliases may be
575 used to define a device which only contains a subset of the resources of
576 the physical device. This can be achieved by only renaming the resources
577 that the device alias should export.
578 -->
579 <alias name="serial_device_1" physical="serial_controller_1">
580   <resource name="ioport1" physical="ioport1"/>
581 </alias>
582 <alias name="nic_1" physical="ethernet_controller_1">
583   <resource name="irq1" physical="irq1">
584     <resource name="msil" physical="msil"/>
585   </resource>
586   <resource name="mem1" physical="mem1"/>
587   <resource name="mem2" physical="mem2"/>
588 </alias>
589 <alias name="storage_controller" physical="sata_controller_1"/>
590 <alias name="ahci_controller" physical="sata_controller_1">
591   <resource name="irq1" physical="irq1">
592     <resource name="msil" physical="msil"/>
593   </resource>
594   <resource name="ahci_registers" physical="mem1"/>
595   <resource name="mmconf" physical="mmconf"/>
596 </alias>
597 </aliases>
598 <classes>
599 <!--
600 The `classes` element specifies a list of device classes.
601 -->
602 <class name="desktop_devices">
603 <!--
604 Device classes enable the grouping of devices and allow referencing all
605 devices by a single name. This simplifies the process of assigning
606 multiple devices to a subject.
607 -->
608 Note: A device class may contain an arbitrary number of devices,
609 including zero.
610 -->
611 <device physical="audio_device_1"/>
612 <device physical="ethernet_controller_1"/>
613 <device physical="network_controller_1"/>
614 <device physical="sata_controller_1"/>
615 <device physical="system_peripheral_1"/>
616 </class>
617 <class name="additional_nics">
618   <device physical="network_controller_1"/>
619 </class>
620 </classes>
621 </mappings>
622 <kernelDiagnostics type="uart">
623 <!--
624 The debug build Muen SK can be instructed to output debugging
625 information during runtime. The platform diagnostics device specifies
626 which device the kernel is to use for this purpose.
627 -->
```

```

629 The presence of this device and the necessary resources are checked by
630 the validator tool.
631 -->
632 <device physical="serial_controller_1">
633   <ioport physical="ioport1"/>
634 </device>
635 </kernelDiagnostics>
636 </platform>
637 <!--
638 This section declares all physical memory regions (RAM) and thus the
639 physical memory layout of the system. Regions declared in this section
640 can be assigned to subjects and device domains.
641 Memory regions are defined by the following attributes:
642
643 - Name
644
645 - Caching type
646
647 - Size
648
649 - Physical address\*
650
651 - Alignment\*
652
653 - Memory type\*
654
655 Attributes with an asterisk are optional. While alignment and memory
656 type are set to a default value if not specified, the physical address
657 is filled in by the allocator tool, which allocates all memory regions
658 and finalizes the physical memory layout.
659
660 Additionally, the content of a region can be declared as backed by a
661 file or filled with a pattern.
662
663 Note: The caching type is an attribute of the physical memory region by
664 design to avoid inconsistent typing, even though the Intel Page
665 Attribute Table (PAT) mechanism allows to set it for each memory
666 mapping, see Intel SDM Vol. 3A, "11.12.4 Programming the PAT".
667 -->
668 <memory caching="WB" name="control_example" size="16#1000#">
669   <fill pattern="16#00#">
670 <!--
671 The 'fill' element designates a memory region which is initialized with
672 the given pattern.
673 -->
674   </fill>
675   <hash value="none"/>
676 </memory>
677 <memory caching="WB" name="control_sm_1" size="16#1000#">
678   <fill pattern="16#00#">
679   <hash value="none"/>
680 </memory>
681 <memory caching="WB" name="control_sm_2" size="16#1000#">
682   <fill pattern="16#ff#">
683   <hash value="none"/>
684 </memory>
685 <memory caching="WB" name="control_time" size="16#1000#">
686   <fill pattern="16#ff#">
687   <hash value="none"/>
688 </memory>
689 <memory caching="WB" name="control_linux_1" size="16#1000#">
690   <fill pattern="16#ff#">
691   <hash value="none"/>
692 </memory>
693 <memory caching="WB" name="status_example" size="16#1000#">
694   <hash value="none"/>
695 </memory>
696 <memory caching="WB" name="status_sm_1" size="16#1000#">
697   <hash value="none"/>
698 </memory>
699 <memory caching="WB" name="status_sm_2" size="16#1000#">
700   <fill pattern="16#00#">
701   <hash value="none"/>
702 </memory>
703 <memory caching="WB" name="status_time" size="16#1000#">
704   <fill pattern="16#00#">
705   <hash value="none"/>
706 </memory>
707 <memory caching="WB" name="status_linux_1" size="16#1000#">
708   <fill pattern="16#00#">
709   <hash value="none"/>
710 </memory>
711 <memory caching="WB" name="slot_control_1" size="16#1000#">
712   <fill pattern="16#00#">
713 </memory>
```

```

715 <memory caching="WB" name="initramfs" size="16#0113_0000#" type="subject_initrd">
716   <file filename="initramfs.cpio.gz" offset="none">
717   <!--
718     The 'file' child element designates a file-backed memory region.
719   --
720     The 'filename' attribute specifies the name of the file to use as
721     content for the physical memory region, the 'offset' attribute is 'none'
722     by default but can be customized to include a partial file.
723   -->
724 </file>
725 </memory>
726 <memory caching="WB" name="nic_linux|ram" size="16#1000_0000#"/>
727 <memory caching="WB" name="nic_linux|lowmem" size="16#0008_0000#"/>
728 <memory caching="WB" name="storage_linux|ram" size="16#1000_0000#"/>
729 <memory caching="WB" name="storage_linux|lowmem" size="16#0008_0000#"/>
730 <memory caching="UC" name="crash_audit" physicalAddress="16#0001_00a1_1000#" size="16#1000#" type="subject_crash_audit"/>
731 </memory>
732 <deviceDomains>
733 <!--
734   The physical memory accessible by PCI devices is specified by so called
735   device domains. Such domains define memory mappings of physical memory
736   regions for one or multiple devices. Device references select a subset
737   of hardware devices provided by the hardware/platform. Devices may be
738   referenced by device name, alias or device class.
739
740   Device references can optionally set the 'mapReservedMemory' attribute
741   so RMRR regions referenced by the device are also mapped into the device
742   domain.
743
744   Device domains are isolated from each other by the use of Intel VT-d.
745   -->
746 <domain name="nic_domain">
747   <memory>
748     <mapSubjectMemory subject="nic_linux"/>
749   </memory>
750   <devices>
751     <device logical="first_nic" physical="ethernet_controller_1"/>
752     <device logical="additional_nics" physical="additional_nics"/>
753   </devices>
754 </domain>
755 <domain name="storage_domain">
756   <memory>
757     <memory executable="false" logical="dma1" physical="storage_linux|lowmem" virtualAddress="16#0002
758       _0000#" writable="true">
759     <!--
760       A 'memory' element maps a physical memory region into the address space
761       of a device domain or subject entity. The region will be accessible to
762       the entity at the specified 'virtualAddress' with permissions defined by
763       the 'executable' and 'writable' attributes.
764     -->
765   </memory>
766   <memory executable="false" logical="dma2" physical="storage_linux|ram" virtualAddress="16#0100_0000#
767     _writable="true"/>
768   </memory>
769   <devices>
770     <device logical="storage_controller" physical="storage_controller"/>
771     <device logical="xhci" physical="usb_controller_1"/>
772   </devices>
773 </domain>
774 </deviceDomains>
775 <events>
776 <!--
777   Events are an activity caused by a subject (source) that impacts a
778   second subject (target) or is directed at the kernel. Events are
779   declared globally and have a unique name to be unambiguous. An event
780   must have a single source and one target.
781
782   Subjects can use events to either deliver an interrupt, hand over
783   execution to or reset the state of a target subject. The first kind of
784   event provides a basic notification mechanism and enables the
785   implementation of event-driven services. The second type facilitates
786   suspension of execution of the source subject and switching to the
787   target. Such a construct is used to pass the thread of execution on to a
788   different subject, e.g. invocation of a debugger subject if an error
789   occurs in the source subject. The third kind is used to facilitate the
790   restart of subjects.
791
792   An event can also have the same source and target, which is called
793   *self* event. Such events are useful to implement para-virtualized
794   timers in VM subjects for example.
795
796   Kernel events are special in that they are targeted at the kernel. The
797   currently supported events are system reboot and shutdown.
798   -->
799 <event mode="switch" name="resume_linux_1">
800 <!--

```

```

799 The 'eventType' specifies an event by name and mode.
801 The following event modes are currently supported:
803 - 'asap'
804 The asap event is an abstraction to state that the event should be
805 delivered as soon as possible, depending on the CPU of the target
806 subject. If the target runs on another CPU core, this mode is
807 expanded to mode *ipi*, which is only available in policy formats A
808 and B, instructing the kernel to preempt the kernel running the
809 target subject and inject the event immediately. If the target
810 subject runs on the same core as the source subject, the mode is
811 expanded to mode *async*.
813 - 'async'
814 Async events trigger no preemption at the target subject. The event
815 is marked as pending in the target subjects pending event table and
816 inserted on the next VM exit/entry cycle of the target subject.
817 - 'self'
818 An event can also have the same source and target, which is called a
819 self event. Such events are useful to implement para-virtualized
820 timers in VM subjects for example. A subject sends itself a delayed
821 event, using the timed event mechanism. Note that a self event must
822 always have a target action assigned, which is checked by the
823 validator.
825 - 'switch'
826 The switch mode facilitates suspension of execution of the source
827 subject and switching to the target. This can only happen between
828 subjects running on the same core. Such a construct is used to pass
829 the thread of execution on to a different subject, e.g. invocation
830 of a debugger subject if an error occurs in the source subject. It
831 is called *handover* or *handover event*.
833 - 'kernel'
834 These kinds of events are directed at the kernel and thus only
835 specify a source since the target is the kernel. They are used to
836 enable specific subjects to unmask level-triggered IRQs and trigger
837 a system reboot, poweroff or explicit panic (crash audit slot
838 allocation and reboot).
-->
841 </event>
842 <event mode="switch" name="resume_linux_2"/>
843 <event mode="switch" name="trap_to_sm_1"/>
844 <event mode="switch" name="trap_to_sm_2"/>
845 <event mode="switch" name="load_linux_1"/>
846 <event mode="switch" name="start_linux_1"/>
847 <event mode="switch" name="reset_linux_1"/>
848 <event mode="switch" name="reset_linux_2"/>
849 <event mode="async" name="reset_sm_1"/>
850 <event mode="async" name="reset_slot_1"/>
851 <event mode="async" name="serial_irq4_linux_1"/>
852 <event mode="async" name="serial_irq4_linux_2"/>
853 <event mode="self" name="timer_linux_1"/>
854 <event mode="self" name="timer_linux_2"/>
855 <event mode="self" name="reboot_linux_1"/>
856 <event mode="kernel" name="system_reboot"/>
857 <event mode="kernel" name="system_poweroff"/>
858 <event mode="kernel" name="system_panic"/>
859 <event mode="self" name="example_self"/>
860 <event mode="kernel" name="subject_sleep"/>
861 <event mode="kernel" name="subject_yield"/>
862 </events>
863 <channels>
<!--
864 Inter-subject communication is specified by so called channels. These
865 channels represent directed information flows since they have a single
866 writer and possibly multiple readers. Optionally a channel can have an
867 associated notification event (doorbell interrupt).
868
869 Channels are declared globally and have an unique name to be
870 unambiguous.
871
872 Note that channels are a policy source format abstraction. The toolchain
873 resolves this concept into memory regions and events as well as the
874 appropriate subject mappings.
-->
875 <channel hasEvent="asap" name="input_events" size="16#1000#">
<!--
876 The 'channel' element declares a physical channel.
877
878 Besides the 'name' and 'size' of the channel, the optional 'hasEvent'
879 attribute can be set to declare that the given channel requests an
880 associated event. The expander tool will then automatically create a
881 global event of the requested event type.
-->

```

```

    </channel>
887 <channel hasEvent="asap" name="virtual_input_1" size="16#1000#"/>
888 <channel hasEvent="asap" name="virtual_input_2" size="16#1000#"/>
889 <channel hasEvent="asap" name="virtual_console_1" size="16#0001_0000#"/>
890 <channel hasEvent="asap" name="virtual_console_2" size="16#0001_0000#"/>
891 <channel name="time_info" size="16#1000#"/>
892 <channel name="debuglog_subject1" size="16#0002_0000#"/>
893 <channel name="debuglog_subject2" size="16#0002_0000#"/>
894 <channel name="debuglog_subject3" size="16#0002_0000#"/>
895 <channel name="debuglog_subject4" size="16#0002_0000#"/>
896 <channel name="debuglog_subject5" size="16#0002_0000#"/>
897 <channel name="debuglog_subject6" size="16#0002_0000#"/>
898 <channel name="debuglog_subject7" size="16#0002_0000#"/>
899 <channel hasEvent="switch" name="nic_dm_request" size="16#1000#"/>
900 <channel hasEvent="switch" name="nic_dm_response" size="16#1000#"/>
901 <channel hasEvent="switch" name="storage_dm_request" size="16#1000#"/>
902 <channel hasEvent="switch" name="storage_dm_response" size="16#1000#"/>
903 <channel name="debuglog_controller" size="16#0002_0000#"/>
904 <channel name="testchannel_1" size="16#1000#"/>
905 <channel name="testchannel_2" size="16#1000#"/>
906 <channel name="testchannel_3" size="16#0010_0000#"/>
907 <channel name="testchannel_4" size="16#0010_0000#"/>
908 <channel name="debuglog_example" size="16#0002_0000#"/>
909 <channel hasEvent="asap" name="example_request" size="16#1000#"/>
910 <channel hasEvent="asap" name="example_response" size="16#1000#"/>
911 </channels>
912 <components>
913 <!--
914   The 'components' element holds a list of components and component
915   libraries.
916
917 Note that components are a policy source format abstraction. The
918 toolchain resolves this concept into subjects by adding the appropriate
919 memory regions, events and devices.
920 -->
921 <library name="libmucontrol">
922 <!--
923 A component library is a specialized component specification which is
924 used to share common resources required for library code to operate.
925 Component libraries can be included by multiple components in order to
926 share functionality. An example is a logging service provided by a
927 dedicated component, whereas the logging client is provided as a library
928 with a shared memory channel for the actual log messages.
929
930 A component specification declares library dependencies to request the
931 library resources from the system through the inclusion of the library
932 specification in the 'depends' section. This way components inherit the
933 resources of libraries.
934
935 On the source code level, a library is included by mechanisms provided
936 by the respective programming language. Note that the component library
937 code is *not* shared between components but lives in the isolated
938 execution environment of a subject instantiating the component (i.e.
939   statically linked libraries).
940
941 Libraries can request the same resources as ordinary components. A
942 subject instantiating the component must also map the resources
943 requested by libraries the component depends on.
944 -->
945 <config/>
946 <requires>
947 <memory>
948 <!--
949 In this section, components can specify expected memory mappings with
950 given access rights and region size.
951 -->
952 <memory executable="false" logical="control" size="16#1000#" virtualAddress="16#000f_ffff_3000#"
953   writable="false">
954 <!--
955 The 'memory' element requests a memory region with the specified 'size'
956 and permissions from the system. The region is expected to be placed at
957 the address given via the 'virtualAddress' attribute.
958 -->
959 </memory>
960 <memory executable="false" logical="status" size="16#1000#" virtualAddress="16#000f_ffff_2000#"
961   writable="true"/>
962 </memory>
963 </requires>
964 </library>
965 <library name="libmudebuglog">
966 <config>
967 <string name="logchannel_size" value="16#0002_0000#"/>
968 </config>
969 <requires>
970 <channels>
971 <!--
972 Components and libraries use the 'channels' sub-section of 'requires' to

```

```

971 specify expected communication channels.
972   -->
973   <writer logical="debuglog" size="16#0002_0000#" virtualAddress="16#000f_fff0_0000#">
974 <!--
975   -->
976   </writer>
977   </channels>
978   </requires>
979   </library>
980   <library name="libmudm">
981     <config/>
982     <requires>
983       <channels>
984         <writer event="8" logical="dm_pciconf_req" size="16#1000#" virtualAddress="16#2000_0000#" />
985         <reader logical="dm_pciconf_res" size="16#1000#" virtualAddress="16#2000_1000#" />
986     <!--
987       The 'reader' element requests a read-only channel of the specified size,
988       address and optional notification vector.
989     -->
990     </reader>
991     </channels>
992     </requires>
993     </library>
994     <library name="libmuinit">
995       <depends>
996         <library ref="libmucontrol"/>
997       </depends>
998     </library>
999     <library name="libmutime">
1000       <config/>
1001       <requires>
1002       <channels>
1003         <reader logical="time_info" size="16#1000#" virtualAddress="16#000f_ffd0_0000#" />
1004       </channels>
1005       </requires>
1006     </library>
1007     <library name="libxhcidbg">
1008       <config/>
1009       <requires>
1010       <memory>
1011         <memory executable="false" logical="xhci_dma" size="16#0004_1000#" virtualAddress="16#0100_0000#"
1012           writable="true"/>
1013       </memory>
1014       <devices>
1015         <device logical="xhci">
1016           <memory executable="false" logical="xhci_registers" size="16#0001_0000#" virtualAddress="16#
1017             e000_0000#" writable="true"/>
1018         </device>
1019       </devices>
1020       </requires>
1021     </library>
1022     <library name="muinit">
1023       <depends>
1024         <library ref="libmuinit"/>
1025       </depends>
1026       <requires>
1027         <vcpu>
1028       <!--
1029         The 'vcpu' element controls the execution behavior of the virtual CPU
1030         (vCPU). A default vCPU profile is selected by the component profile, but
1031         CPU execution settings can be customized both at component and subject
1032         level.
1033       -->
1034       <registers>
1035         <gpr>
1036           <rip>16#0010_0000#</rip>
1037         </gpr>
1038       </registers>
1039     </vcpu>
1040     </requires>
1041     <provides>
1042       <memory executable="true" logical="muinit" size="16#9000#" type="subject_binary" virtualAddress="
1043         16#0010_0000#" writable="false">
1044         <file filename="muinit" offset="none"/>
1045       </memory>
1046     </provides>
1047     </library>
1048     <component name="ahci_drv" profile="native">
1049       <!--
1050         A component is a piece of software which shall be executed by the SK.
1051         Components represent the building blocks of a component-based system and
1052         can be regarded as templates for executable entities instantiated by
1053         subjects.
1054       The specification of a component declares the *binary program* by means
1055         of (file-backed memory) regions. It also specifies the components view

```

```

1055 of the expected execution environment. A component may request the
1057 following resources from the system:
1059 - Logical channels
1061 - Logical memory regions
1063 - Logical devices
1065 - Logical events
1067 Components are identified by name and specify a profile. The profile
1068 controls the settings of the virtual CPU (vCPU).
-->
1069 <config>
<!--
1071 Components may declare their own configuration values in the config
section. Just like global system config values, these can also be
1073 used in '<if>' expressions and XML attribute value expansion.
-->
1075 </config>
<depends>
1077 <!--
1078 Components and libraries are allowed to declare dependencies to other
1079 libraries. All resources required by the included library are merged
1080 with the ones specified by the component or library. Libraries can
1081 depend on other libraries.
1083 A subject realizing this component must correctly map all component and
library resource requirements to physical resources in order to fulfill
1085 the expectations.
-->
1087 <library ref="libmudebuglog"/>
</depends>
1089 <requires>
<vcpu>
1091 <registers>
<gpr>
1093 <rip>16#0020_0000#</rip>
</gpr>
1095 </registers>
</vcpu>
1097 <memory>
<!-- for 32 ports 16#c000# bytes are needed for descriptor tables + 16K for device init -->
1099 <memory executable="false" logical="dma_region" size="16#0001_0000#" virtualAddress="16#a000_0000#"
      writable="true"/>
<array elementSize="16#0100_0000#" executable="false" logical="blockdev_shm" virtualAddressBase="16#
      a100_0000#" writable="true">
<memory logical="blockdev_shm1"/>
<memory logical="blockdev_shm2"/>
1103 </array>
</memory>
<channels>
<array elementSize="16#0000_8000#" logical="blockdev_request" vectorBase="50" virtualAddressBase="
      16#0200_0000#">
1107 <!--
1108   The channel array abstraction simplifies the declaration of consecutive
1109   channel mappings with a given base address, channel size and optional
1110   event/vector bases. The child elements declare the number of expected
1111   channels and either the 'reader' or 'writer' role.
-->
1113 <reader logical="blockdev_request1">
<!--
1115   Array entries specify the number of array elements and assign a logical
1116   name to each element.
-->
1117 </reader>
1119 <reader logical="blockdev_request2"/>
</array>
1121 <array elementSize="16#0000_4000#" eventBase="40" logical="blockdev_response" virtualAddressBase="
      16#0300_0000#">
<writer logical="blockdev_responsel"/>
<writer logical="blockdev_response2"/>
1125 </array>
</channels>
<devices>
1127 <!--
1128   The 'devices' sub-section of the 'requires' section is used to specify
1129   expected devices with their associated resources.
-->
1131 <device logical="ahci_controller">
<!--
1133   A 'device' element specifies an expected logical device with its
resources. Possible resources are 'irq', 'memory' and 'ioPort'.
-->
1135 <irq logical="irq" vector="48">
<!--

```

```

1139     An 'irq' element of a logical device reference requests an IRQ with
1140     given number from the system policy. The specified number will be
1141     injected when the device requires attention for the associated logical
1142     function.
1143     -->
1144     </irq>
1145     <memory executable="false" logical="ahci_registers" size="16#1000#" virtualAddress="16#e000_0000#"
1146         writable="true"/>
1147     <memory executable="false" logical="mmconf" size="16#1000#" virtualAddress="16#f800_8000#" writable=
1148         "true"/>
1149     </device>
1150     </devices>
1151     </requires>
1152     <provides>
1153     <!--
1154     Components usually come in the form of an executable file. To this end,
1155     the 'provides' section specifies the memory regions of the component
1156     binary executable with their content.
1157
1158     From a security perspective, it is often desirable to provide the
1159     different binary section as separate memory regions with the appropriate
1160     access rights, i.e. only the text section is executable, rodata is not
1161     writable and so on.
1162
1163     Memory specified in this sections are expanded to mapped physical
1164     regions for each subject that instantiates this component.
1165
1166     Note: the Mucbinsplit tool can be used to extract these section from an
1167     ELF binary into separate files and automatically add the corresponding
1168     memory elements to the component specification.
1169     -->
1170     <memory executable="true" logical="text" size="16#9000#" type="subject_binary" virtualAddress="
1171         16#0020_0000#" writable="false">
1172     <!--
1173     A 'memory' element in the 'provides' section declares memory region
1174     provided by the component. Mostly used to provide (a part) of the
1175     component binary.
1176     -->
1177     <file filename="ahci_drv_text" offset="none"/>
1178     <hash value="16#607269d054db8d50977cf503b9629c1ec05083949fe3ea3787c7968a7c5be4bc#" />
1179     </memory>
1180     <memory executable="false" logical="rodata" size="16#1000#" type="subject_binary" virtualAddress="
1181         16#0020_c000#" writable="false">
1182     <file filename="ahci_drv_rodata" offset="none"/>
1183     <hash value="16#f7eba385de0bba39cc5d5e860fca26c765ed0fd05b8ba326639e5c699341251a#" />
1184     </memory>
1185     <memory executable="false" logical="data" size="16#1000#" type="subject_binary" virtualAddress="
1186         16#0020_d000#" writable="true">
1187     <file filename="ahci_drv_data" offset="none"/>
1188     <hash value="16#1cb2148aeef42e097dad59630d8940dce44f8b351f88fcfac9ded03d0a2a831b1#" />
1189     </memory>
1190     <memory executable="false" logical="bss" size="16#3000#" type="subject_binary" virtualAddress="
1191         16#0020_9000#" writable="true">
1192     <fill pattern="16#00#" />
1193     </memory>
1194     <memory executable="false" logical="stack" size="16#4000#" type="subject_binary" virtualAddress="
1195         16#1000#" writable="true">
1196     <fill pattern="16#00#" />
1197     </memory>
1198     </provides>
1199     </component>
1200     <component name="controller" profile="native">
1201     <config/>
1202     <depends>
1203     <library ref="libmudebuglog"/>
1204     </depends>
1205     <requires>
1206     <vcpu>
1207     <registers>
1208     <gpr>
1209     <rip>16#0020_0000#</rip>
1210     </gpr>
1211     </registers>
1212     </vcpu>
1213     <memory>
1214     <array elementSize="16#1000#" executable="false" logical="control" virtualAddressBase="16#a000_0000#
1215         " writable="true">
1216     <memory logical="control_1"/>
1217     <memory logical="control_2"/>
1218     <memory logical="control_3"/>
1219     <memory logical="control_4"/>
1220     <memory logical="control_5"/>
1221     </array>
1222     <array elementSize="16#1000#" executable="false" logical="status" virtualAddressBase="16#a100_0000#
1223         " writable="false">
1224     <memory logical="status_1"/>
1225     <memory logical="status_2"/>

```

```

1217   <memory logical="status_3"/>
1218   <memory logical="status_4"/>
1219   <memory logical="status_5"/>
1220 </array>
1221 <memory executable="false" logical="slot_control_1" size="16#1000#" virtualAddress="16#9000_0000#"
1222   writable="false"/>
1223 </memory>
1224 <events>
1225 <!--
1226 The 'events' sub-section of the 'requires' section is used to specify
1227 expected events with optional event actions.
1228
1229 A component can specify both source as well as target events.
1230 -->
1231 <source>
1232   <event id="10" logical="reset_slot_1_sm"/>
1233   <event id="11" logical="reset_slot_1_linux"/>
1234 </source>
1235 </events>
1236 </requires>
1237 <provides>
1238   <memory executable="true" logical="text" size="16#2000#" type="subject_binary" virtualAddress="16#0020_0000#"
1239     writable="false">
1240     <file filename="controller_text" offset="none"/>
1241     <hash value="16#631dbc08c3c5970c10bab7086be84c0ec70d79330f538751a27f18eaf8d4441#"/>
1242   </memory>
1243   <memory executable="false" logical="rodata" size="16#1000#" type="subject_binary" virtualAddress="16#0020_3000#"
1244     writable="false">
1245     <file filename="controller_rodata" offset="none"/>
1246     <hash value="16#df41e836ab51453074f093d2824347d90d272ec8abaf4197c5ddb7636fd83320#"/>
1247   </memory>
1248   <memory executable="false" logical="data" size="16#1000#" type="subject_binary" virtualAddress="16#0020_4000#"
1249     writable="true">
1250     <file filename="controller_data" offset="none"/>
1251     <hash value="16#ad7facb2586fc6e966c004d7d1d16b024ff5805ff7cb47c7a85dabd8b48892ca7#"/>
1252   </memory>
1253   <memory executable="false" logical="bss" size="16#1000#" type="subject_binary" virtualAddress="16#0020_2000#"
1254     writable="true">
1255     <fill pattern="16#00#"/>
1256   </memory>
1257   <memory executable="false" logical="stack" size="16#4000#" type="subject_binary" virtualAddress="16#1000#"
1258     writable="true">
1259     <fill pattern="16#00#"/>
1260   </memory>
1261 </provides>
1262 </component>
1263 <component name="dbgserver" profile="native">
1264   <config>
1265     <boolean name="sink_serial" value="true"/>
1266     <boolean name="sink_shmem" value="false"/>
1267     <boolean name="hsuart_enabled" value="false"/>
1268     <boolean name="sink_xhcidbg" value="false"/>
1269     <boolean name="default_channel_enabled_state" value="true"/>
1270     <boolean name="sink_pcspkr" value="false"/>
1271     <string name="logchannel_size" value="16#0002_0000#"/>
1272     <string name="debugconsole_port_start" value="16#60b0#"/>
1273     <string name="enabled_channels_override" value="" />
1274     <string name="debugconsole_port_end" value="16#60b7#"/>
1275   </config>
1276   <requires>
1277     <vcpu>
1278       <registers>
1279         <gpr>
1280           <rip>16#0020_0000</rip>
1281         </gpr>
1282       </registers>
1283     </vcpu>
1284   <memory>
1285     <memory executable="false" logical="crash_audit" size="16#1000#" virtualAddress="16#1000_0000#"
1286       writable="false"/>
1287   </memory>
1288   <channels>
1289     <array elementSize="16#0002_0000#" logical="log_channels" virtualAddressBase="16#a000_0000#">
1290       <reader logical="log_channel1"/>
1291       <reader logical="log_channel2"/>
1292       <reader logical="log_channel3"/>
1293       <reader logical="log_channel4"/>
1294       <reader logical="log_channel5"/>
1295       <reader logical="log_channel_example"/>
1296       <reader logical="log_channel_6"/>
1297       <reader logical="log_channel7"/>
1298       <reader logical="log_channel8"/>
1299     </array>
1300   </channels>
1301   <devices>
1302     <device logical="debugconsole">
1303       <ioPort end="16#60b7#" logical="port" start="16#60b0#">

```

```

<!--
1297   The 'ioPort' element requests a device I/O port resource with given
1298   range 'start .. end' from the system.
1299   -->
1300   </ioPort>
1301   </device>
1302   </devices>
1303   <events>
1304   <source>
1305     <event id="30" logical="shutdown">
1306       <system_poweroff>
1307     <!--
1308       An example of a source event action directed at the kernel. If this
1309       event is triggered by the associated subject, the system will power
1310       off.
1311     -->
1312     </system_poweroff>
1313   </event>
1314   <event id="31" logical="reboot">
1315     <system_reboot/>
1316   </event>
1317   </source>
1318   </events>
1319   </requires>
1320   <provides>
1321     <memory executable="true" logical="text" size="16#a000#" type="subject_binary" virtualAddress="
1322       16#0020_0000#" writable="false">
1323       <file filename="dbgserver_text" offset="none"/>
1324     <hash value="16#f0904b10d5621c49e1lab5f5fb25e1f22c63071bf19ae1eef2cbb57a0c87d17#/"/>
1325   </memory>
1326   <memory executable="false" logical="rodata" size="16#1000#" type="subject_binary" virtualAddress="
1327     16#0021_8000#" writable="false">
1328     <file filename="dbgserver_rodata" offset="none"/>
1329     <hash value="16#f5ad233b4e8da945aa5fc73c256f2a5bfb904c1b6b6d4b6dd277052cd0513c62#/"/>
1330   </memory>
1331   <memory executable="false" logical="data" size="16#1000#" type="subject_binary" virtualAddress="
1332     16#0021_9000#" writable="true">
1333     <file filename="dbgserver_data" offset="none"/>
1334     <hash value="16#0f34e786f747c7d50777202a76bf0015016d64ca7dc2583a68aa301e32d0b632#/"/>
1335   </memory>
1336   <memory executable="false" logical="bss" size="16#e000#" type="subject_binary" virtualAddress="
1337     16#0020_a000#" writable="true">
1338     <fill pattern="16#00#"/>
1339   </memory>
1340   <memory executable="false" logical="stack" size="16#4000#" type="subject_binary" virtualAddress="
1341     16#1000#" writable="true">
1342     <fill pattern="16#00#"/>
1343   </memory>
1344   </provides>
1345   </component>
1346   <component name="dm" profile="native">
1347     <config/>
1348     <depends>
1349       <library ref="libmudebuglog"/>
1350     </depends>
1351     <requires>
1352       <vcpu>
1353         <registers>
1354           <gpr>
1355             <rip>16#0020_0000</rip>
1356           </gpr>
1357         </registers>
1358       </vcpu>
1359       <channels>
1360         <reader logical="request" size="16#1000#" virtualAddress="16#0010_0000#/"/>
1361         <writer event="9" logical="response" size="16#1000#" virtualAddress="16#0010_1000#/"/>
1362       </channels>
1363     </requires>
1364     <provides>
1365       <memory executable="true" logical="text" size="16#4000#" type="subject_binary" virtualAddress="
1366         16#0020_0000#" writable="false">
1367         <file filename="dm_text" offset="none"/>
1368         <hash value="16#1eb87d2f399da0e0b924bef55a0ad95388c1c4ecalf8cd4c7448f5247eefd3d#/"/>
1369       </memory>
1370       <memory executable="false" logical="rodata" size="16#1000#" type="subject_binary" virtualAddress="
1371         16#0020_5000#" writable="false">
1372         <file filename="dm_rodata" offset="none"/>
1373         <hash value="16#16618d141be7aa0818730c944df3146b185b467f39c1a2163074c4af1d3e8894#/"/>
1374       </memory>
1375       <memory executable="false" logical="data" size="16#1000#" type="subject_binary" virtualAddress="
1376         16#0020_6000#" writable="true">
1377         <file filename="dm_data" offset="none"/>
1378         <hash value="16#9348d0aec58fa2e80f74bf3a52440ec842aa7e6bf948bcd98d40d1c30dc218ec#/"/>
1379       </memory>
1380       <memory executable="false" logical="bss" size="16#1000#" type="subject_binary" virtualAddress="
1381         16#0020_4000#" writable="true">
1382         <fill pattern="16#00#"/>
1383       </memory>

```

```

    </memory>
1375 <memory executable="false" logical="stack" size="16#4000#" type="subject_binary" virtualAddress="16#1000#" writable="true">
    <fill pattern="16#00#" />
</memory>
</provides>
</component>
<component name="example" profile="native">
<config>
    <boolean name="ahci_drv_enabled" value="false"/>
1383 <boolean name="print_serial" value="false"/>
    <boolean name="print_vcpu_speed" value="true"/>
1385 <integer name="serial" value="123456789"/>
    <string name="greeter" value="Subject running"/>
</config>
<depends>
1389 <library ref="libmudebuglog"/>
    <library ref="muinit"/>
</depends>
<requires>
1393 <vcpu>
    <vmx>
1395 <masks>
        <exception>
            <Breakpoint>0</Breakpoint>
        </exception>
        </masks>
    </vmx>
1401 <registers>
    <gpr>
        <rip>16#0020_0000#</rip>
    </gpr>
</registers>
</vcpu>
1407 <memory>
    </memory>
<channels>
    <reader logical="example_request" size="16#1000#" vector="42" virtualAddress="16#000A_0000#"/>
1411 <writer event="1" logical="example_response" size="16#1000#" virtualAddress="16#000B_0000#"/>
</channels>
1413 <events>
    <source>
        <event id="2" logical="yield">
            <subject_yield/>
        </event>
        <event id="3" logical="timer">
            <event id="4" logical="sleep">
                <subject_sleep/>
            </event>
        </source>
        <target>
            <event logical="inject_timer">
                <inject_interrupt vector="37"/>
            </event>
        </target>
    </events>
1423 <requires>
    <provides>
1431 <memory executable="true" logical="text" size="16#4000#" type="subject_binary" virtualAddress="16#0020_0000#" writable="false">
        <file filename="example_text" offset="none"/>
1433 <hash value="16#72b95e0ab8ffa96d0ddad39b4becd018e2ea221e49ba1305b7dee07b36e624cf#"/>
</memory>
1435 <memory executable="false" logical="rodata" size="16#1000#" type="subject_binary" virtualAddress="16#0020_6000#" writable="false">
        <file filename="example_rodata" offset="none"/>
1437 <hash value="16#8c62e10ab457e3d56f2defc177c76879dal4c6262ffe488ccfc064cc613a08e3#"/>
</memory>
1439 <memory executable="false" logical="data" size="16#1000#" type="subject_binary" virtualAddress="16#0020_7000#" writable="true">
        <file filename="example_data" offset="none"/>
1441 <hash value="16#f738ffff901c6481a7c8ec777bb83120a0bcf6bf072fd0ea8198a876af5c7ef1b#"/>
</memory>
1443 <memory executable="false" logical="bss" size="16#2000#" type="subject_binary" virtualAddress="16#0020_4000#" writable="true">
        <fill pattern="16#00#"/>
</memory>
1445 <memory executable="false" logical="stack" size="16#4000#" type="subject_binary" virtualAddress="16#1000#" writable="true">
        <fill pattern="16#00#"/>
</memory>
</provides>
</component>
<component name="idle" profile="native">
    <requires>
        <vcpu>
            <registers>

```

```

1455 <gpr>
1456   <rip>16#0020_0000#</rip>
1457 </gpr>
1458 </registers>
1459 </vcpu>
1460 </requires>
1461 <provides>
1462   <memory executable="true" logical="text" size="16#1000#" type="subject_binary" virtualAddress="16#0020_0000#" writable="false">
1463     <file filename="idle_text" offset="none"/>
1464     <hash value="16#3f85e2e49adb66104e3292d306517c5581cd7c226ba60e66a4414269083f8e8d#"/>
1465   </memory>
1466   <memory executable="false" logical="rodata" size="16#1000#" type="subject_binary" virtualAddress="16#0020_1000#" writable="false">
1467     <file filename="idle_rodata" offset="none"/>
1468     <hash value="16#db97c62b590d580647fe04fcc6c8a962697fa51f0a7ab475a16967e29cbb4cb9#"/>
1469   </memory>
1470   <memory executable="false" logical="stack" size="16#4000#" type="subject_binary" virtualAddress="16#1000#" writable="true">
1471     <fill pattern="16#00#"/>
1472   </memory>
1473 </provides>
1474 </component>
1475 <component name="isolation_tests_monitor" profile="native">
1476   <config/>
1477   <requires>
1478     <vcpu>
1479       <registers>
1480         <gpr>
1481           <rip>16#0020_0000#</rip>
1482         </gpr>
1483       </registers>
1484     </vcpu>
1485     <memory>
1486       <memory executable="false" logical="result_state" size="16#1000#" virtualAddress="16#0100_0000#" writable="true"/>
1487     </memory>
1488     <events>
1489       <source>
1490         <event id="1" logical="resume_tests"/>
1491       </source>
1492       <target>
1493         <event logical="trap_to_monitor"/>
1494       </target>
1495     </events>
1496   </requires>
1497   <provides>
1498     <memory executable="true" logical="text" size="16#1000#" type="subject_binary" virtualAddress="16#0020_0000#" writable="false">
1499       <file filename="isolation_tests_monitor_text" offset="none"/>
1500       <hash value="16#7feb0756df521a4a652c42169ef4d4f0c665aef9bb29c09f5f55969baf718fa7#"/>
1501     </memory>
1502     <memory executable="false" logical="rodata" size="16#1000#" type="subject_binary" virtualAddress="16#0020_1000#" writable="false">
1503       <file filename="isolation_tests_monitor_rodata" offset="none"/>
1504       <hash value="16#dd605fb441991ff03476a906d81d12420eeeb511a501874bcdca0a761a311657#"/>
1505     </memory>
1506     <memory executable="false" logical="data" size="16#1000#" type="subject_binary" virtualAddress="16#0020_2000#" writable="true">
1507       <file filename="isolation_tests_monitor_data" offset="none"/>
1508       <hash value="16#ad7facb2586fc6e966c004d7d1d16b024f5805ff7cb47c7a85dabd8b48892ca7#"/>
1509     </memory>
1510     <memory executable="false" logical="stack" size="16#4000#" type="subject_binary" virtualAddress="16#1000#" writable="true">
1511       <fill pattern="16#00#"/>
1512     </memory>
1513   </provides>
1514 </component>
1515 <component name="isolation_tests" profile="native">
1516   <config>
1517     <integer name="log_entry_max" value="128"/>
1518     <integer name="log_buffer_size" value="65535"/>
1519   </config>
1520   <depends>
1521     <library ref="libmudebuglog"/>
1522   </depends>
1523   <requires>
1524     <vcpu>
1525       <msrs>
1526         <msr end="16#0174#" mode="r" start="16#0174#"/>
1527       </msrs>
1528       <registers>
1529         <gpr>
1530           <rip>16#0020_0000#</rip>
1531         </gpr>
1532       </registers>
1533     </vcpu>

```

```

<memory>
1535 <memory executable="false" logical="read_only" size="16#1000#" virtualAddress="16#1000_0000#"
      writable="false"/>
<memory executable="false" logical="result_state" size="16#1000#" virtualAddress="16#0100_0000#"
      writable="true"/>
1537 </memory>
<events>
1539 <target>
<event logical="resume_tests"/>
1541 </target>
<events>
1543 </requires>
<provides>
1545 <memory executable="true" logical="text" size="16#7000#" type="subject_binary" virtualAddress="
      16#0020_0000#" writable="false">
<file filename="isolation_tests_text" offset="none"/>
1547 <hash value="16#8a97f6a71450066d433b34a76442771fb5d8feab69471819fc68729a02dac92b#" />
</memory>
1549 <memory executable="false" logical="rodata" size="16#3000#" type="subject_binary" virtualAddress="
      16#0021_8000#" writable="false">
<file filename="isolation_tests_rodata" offset="none"/>
1551 <hash value="16#29a8a7797c28bcd3d394fbc205e3ff06b598fa1838edfb4cbed9086259c071#" />
</memory>
1553 <memory executable="false" logical="data" size="16#b000#" type="subject_binary" virtualAddress="
      16#0021_b000#" writable="true">
<file filename="isolation_tests_data" offset="none"/>
1555 <hash value="16#731b8abc3033235953e353ac5e0d68f1611e9739edb0e1ef274a0403ce883bef#" />
</memory>
1557 <memory executable="false" logical="bss" size="16#0001_1000#" type="subject_binary" virtualAddress="
      16#0020_7000#" writable="true">
<fill pattern="16#00#" />
1559 </memory>
<memory executable="false" logical="stack" size="16#4000#" type="subject_binary" virtualAddress="
      16#1000#" writable="true">
<fill pattern="16#00#" />
1561 </memory>
</provides>
</component>
1563 <component name="linux" profile="linux">
<requires>
1565 <memory>
<memory executable="true" logical="lowmem" size="16#0008_0000#" virtualAddress="16#0002_0000#"
      writable="true"/>
1569 <memory executable="true" logical="ram" size="16#1000_0000#" virtualAddress="16#0100_0000#" writable
      ="true"/>
</memory>
1571 </requires>
<provides>
1573 <memory executable="true" logical="binary" size="16#0062_8000#" type="subject_binary" virtualAddress
      ="16#0040_0000#" writable="true">
<file filename="bzImage" offset="none"/>
1575 </memory>
<memory executable="false" logical="modules_initramfs" size="16#0001_f000#" type="subject_initrd"
      virtualAddress="16#7113_0000#" writable="false">
<file filename="modules_initramfs.cpio.gz" offset="none"/>
1577 </memory>
</provides>
</component>
1581 <component name="ps2_drv" profile="native">
<config/>
1583 <depends>
<library ref="libmudebuglog"/>
1585 </depends>
<requires>
1587 <vcpu>
<registers>
1589 <rip>16#0020_0000</rip>
1591 </gpr>
</registers>
1593 </vcpu>
<channels>
1595 <writer event="1" logical="input_events" size="16#1000#" virtualAddress="16#0005_0000#" />
</channels>
1597 <devices>
<device logical="ps2">
1599 <irq logical="kbd_irq" vector="49"/>
<irq logical="mouse_irq" vector="60"/>
1601 <ioPort end="16#0060#" logical="port_60" start="16#0060#/"/>
<ioPort end="16#0064#" logical="port_64" start="16#0064#/"/>
1603 </device>
</devices>
1605 </requires>
<provides>
1607 <memory executable="true" logical="text" size="16#3000#" type="subject_binary" virtualAddress="
      16#0020_0000#" writable="false">
<file filename="ps2_drv_text" offset="none"/>

```

```

1609 <hash value="16#884d113adb35a8e7206fdbd357731deb78cc18d849a95a1add1c875f7c24971b#" />
1610 </memory>
1611 <memory executable="false" logical="rodata" size="16#1000#" type="subject_binary" virtualAddress="16#0020_5000#" writable="false">
1612   <file filename="ps2_drv_rodata" offset="none"/>
1613   <hash value="16#232bd0e2a4f9b4f51be60d6617513b5db259105824a15d4e42ef5a63561f307#" />
1614 </memory>
1615 <memory executable="false" logical="data" size="16#1000#" type="subject_binary" virtualAddress="16#0020_6000#" writable="true">
1616   <file filename="ps2_drv_data" offset="none"/>
1617   <hash value="16#442d8d89919baff84d8a3947efb6c995901fc2539f475d8bba0913e4cca8f683#" />
1618 </memory>
1619 <memory executable="false" logical="bss" size="16#2000#" type="subject_binary" virtualAddress="16#0020_3000#" writable="true">
1620   <fill pattern="16#00#" />
1621 </memory>
1622 <memory executable="false" logical="stack" size="16#4000#" type="subject_binary" virtualAddress="16#1000#" writable="true">
1623   <fill pattern="16#00#" />
1624 </memory>
1625 </provides>
1626 </component>
1627 <component name="sl" profile="native">
1628   <config/>
1629   <depends>
1630     <library ref="libmuinit"/>
1631   </depends>
1632   <requires>
1633     <vcpu>
1634       <registers>
1635         <gpr>
1636           <rip>16#0020_0000#</rip>
1637         </gpr>
1638       </registers>
1639     </vcpu>
1640     <events>
1641       <source>
1642         <event id="0" logical="start"/>
1643       </source>
1644       <target>
1645         <event logical="handle_reset"/>
1646       </target>
1647     </events>
1648   </requires>
1649   <provides>
1650     <memory executable="true" logical="text" size="16#3000#" type="subject_binary" virtualAddress="16#0020_0000#" writable="false">
1651       <file filename="sl_text" offset="none"/>
1652       <hash value="16#8a939bec2bb65fc92483662ac3cccc33df668fbef8ba82e702e8bba8525d299a#" />
1653     </memory>
1654     <memory executable="false" logical="rodata" size="16#1000#" type="subject_binary" virtualAddress="16#0020_3000#" writable="false">
1655       <file filename="sl_rodata" offset="none"/>
1656       <hash value="16#680bc1a5bea402fb9232ce73620ece595c32696b86c6b9c4f3eb159cbd270e31#" />
1657     </memory>
1658     <memory executable="false" logical="data" size="16#1000#" type="subject_binary" virtualAddress="16#0020_4000#" writable="true">
1659       <file filename="sl_data" offset="none"/>
1660       <hash value="none"/>
1661     </memory>
1662     <memory executable="false" logical="stack" size="16#4000#" type="subject_binary" virtualAddress="16#1000#" writable="true">
1663       <fill pattern="16#00#" />
1664     </memory>
1665   </provides>
1666 </component>
1667 <component name="sm" profile="native">
1668   <config>
1669     <boolean name="debug_wrmsr" value="false"/>
1670     <boolean name="pciconf_emulation_enabled" value="true"/>
1671     <boolean name="debug_rdtsc" value="false"/>
1672     <boolean name="debug_rdmsr" value="false"/>
1673     <boolean name="debug_cpuid" value="false"/>
1674     <boolean name="debug_cr" value="false"/>
1675     <boolean name="debug_ioport" value="false"/>
1676     <boolean name="debug_ept" value="false"/>
1677   </config>
1678   <depends>
1679     <library ref="libmutime"/>
1680     <library ref="libmudebuglog"/>
1681     <library ref="libmudm"/>
1682     <library ref="muinit"/>
1683   </depends>
1684   <requires>
1685     <vcpu>
1686     <registers>
1687       <gpr>

```

```

1689 <rip>16#0020_0000#</rip>
1690 </gpr>
1691 </registers>
1692 </vcpu>
1693 <events>
1694 <event id="4" logical="resume_subject"/>
1695 </source>
1696 <target>
1697 <event logical="handle_subject_trap"/>
1698 </target>
1699 </events>
1700 </requires>
1701 <provides>
1702 <memory executable="true" logical="text" size="16#6000#" type="subject_binary" virtualAddress="16#0020_0000#" writable="false">
1703 <file filename="sm_text" offset="none"/>
1704 <hash value="16#d7963866a17ee6a4fac490d6bb536c84475dcd80377f3c84bf8af8769697f196#/"/>
1705 </memory>
1706 <memory executable="false" logical="rodata" size="16#1000#" type="subject_binary" virtualAddress="16#0020_8000#" writable="false">
1707 <file filename="sm_rodata" offset="none"/>
1708 <hash value="16#80b22f10ab78e493997b23748b0e44aeb58e17fa5f8fb3936ab591055df2763b#/"/>
1709 </memory>
1710 <memory executable="false" logical="data" size="16#1000#" type="subject_binary" virtualAddress="16#0020_9000#" writable="true">
1711 <file filename="sm_data" offset="none"/>
1712 <hash value="16#cbd2cd8430f377f1dea2537c50ece5c784d635812b7c4fc25602d77ab3c1b2c1#/"/>
1713 </memory>
1714 <memory executable="false" logical="bss" size="16#2000#" type="subject_binary" virtualAddress="16#0020_6000#" writable="true">
1715 <fill pattern="16#00#/"/>
1716 </memory>
1717 <memory executable="false" logical="stack" size="16#4000#" type="subject_binary" virtualAddress="16#1000#" writable="true">
1718 <fill pattern="16#00#/"/>
1719 </memory>
1720 </provides>
1721 </component>
1722 <component name="time" profile="native">
1723 <config/>
1724 <depends>
1725 <library ref="libmudebuglog"/>
1726 <library ref="libmucontrol"/>
1727 </depends>
1728 <requires>
1729 <vcpu>
1730 <vmx>
1731 <controls>
1732 <proc>
1733 <RDTSCExit>0<!!--
This is an example of a component that customizes the vCPU
settings. In this case, direct access to the Time-Stamp Counter
(TSC) is enabled. The settings made here are merged with the
(default) values defined by the component profile during policy
expansion by the Mucfgexpand tool.
--></RDTSCExit>
1734 </proc>
1735 </controls>
1736 </vmx>
1737 <registers>
1738 <gpr>
1739 <rip>16#0020_0000#</rip>
1740 </gpr>
1741 </registers>
1742 </vcpu>
1743 <channels>
1744 <array elementSize="16#1000#" logical="export_channels" virtualAddressBase="16#000f_ffd0_0000#/"/>
1745 <writer logical="time_export1"/>
1746 </array>
1747 </channels>
1748 <devices>
1749 <device logical="cmos_rtc">
1750 <ioPort end="16#0071#" logical="ports" start="16#0070#/"/>
1751 </device>
1752 </devices>
1753 </requires>
1754 <provides>
1755 <memory executable="true" logical="text" size="16#2000#" type="subject_binary" virtualAddress="16#0020_0000#" writable="false">
1756 <file filename="time_text" offset="none"/>
1757 <hash value="16#75f74e5bdae421c5bc86daa0749ebd404a0a800e3042f963bbe5d774f25d33a8#/"/>
1758 </memory>
1759 <memory executable="false" logical="rodata" size="16#1000#" type="subject_binary" virtualAddress="16#0020_3000#" writable="false">
1760 <file filename="time_rodata" offset="none"/>
1761 <hash value="16#b0fd849e1cb2a24c297924bebe30b0d11e47258ded43364b726431e322a233b6#/"/>

```

```

        
```

```

1843  </memory>
1844  </provides>
1845  </component>
1846  </components>
1847  <subjects>
<!--
1849  The 'subjects' element holds a list of subjects.
-->
1851  <subject name="vt">
<!--
1853  A subject is an instance of a component, i.e. an active component in the
1854  system policy that may be scheduled. Its specification references a
1855  component and maps all requested logical resources to physical resources
1856  provided by the system. Additional resources to the ones requested by
1857  the component can be specified here. This enables specialization of the
1858  base component specification.
-->
1859  <events>
<!--
1861  The subject 'events' element specifies all events originating from or
1862  directed at this subject. The physical attribute is the name of a event
1863  defined in the global events section.
-->
1865  <source>
<!--
1867  The event 'source' element specifies events that are allowed to be
1868  triggered by the associated subject.
-->
1871  The 'vmx_exit' group is translated to a lookup table for handling VMX
1872  exit traps as defined by Intel SDM Vol. 3D, "Appendix C VMX Basic Exit
1873  Reasons". The 'vmcall' group on the other hand is translated into a
1874  lookup table to handle hypercalls.
-->
1875  <group name="vmx_exit">
1876    <default physical="system_panic">
<!--
1879  The 'default' element entry can be used to specify an event which should
1880  be added for all event ids that have not been explicitly specified.
-->
1881  <system_panic/>
</default>
1883  </group>
1885  </source>
1886  </events>
1887  <component ref="vt">
<!--
1889  The 'component' reference element specifies which component this subject
1890  instantiates. All logical resources required by the component must be
1891  mapped to physical resources of the appropriate type. Validators make
1892  sure that all requirements are satisfied and that no mapping has been
1893  omitted.
-->
1895  <map logical="NIC_Linux" physical="virtual_console_1"/>
1896  <map logical="Storage_Linux" physical="virtual_console_2"/>
1897  <map logical="input_events" physical="input_events"/>
1898  <map logical="input_device_1" physical="virtual_input_1"/>
1899  <map logical="input_device_2" physical="virtual_input_2"/>
1900  <map logical="debuglog" physical="debuglog_subject1"/>
1901  <map logical="vga" physical="vga">
1902  <map logical="buffer" physical="buffer"/>
1903  <map logical="ports" physical="ports">
<!--
1905  The 'map' element maps a physical resource provided by the system with a
1906  resource requested by the referenced component.
-->
1907  This element allows recursion to map child resources as well (e.g.
1908    device memory, I/O ports etc.
-->
1909  </map>
</map>
1910  <map logical="slot_control_1" physical="slot_control_1"/>
1911  <map logical="shutdown" physical="system_poweroff"/>
1912  <map logical="reboot" physical="system_reboot"/>
</component>
</subject>
<subject name="nic_sm">
1919  <memory executable="false" logical="status_linux" physical="status_linux_1" virtualAddress="16#0200
1920    _0000#" writable="false"/>
</memory>
<events>
<source>
1923  <group name="vmx_exit">
1924    <default physical="system_panic">
1925      <system_panic/>
</default>
</group>

```

```

1929 <group name="vmcall">
1930   <event id="0" logical="serial_irq4" physical="serial_irq4_linux_1">
1931   <!--
1932     A source 'event' entry specifies a source event node, i.e. it registers
1933     a handler for the given event 'id'. These IDs, depending on the event
1934     group, are either hypercall numbers or VMX basic exit reasons. The valid
1935     ID ranges of the respective groups are:
1936
1937   - vmx\_exit
1938   0 .. 59
1939
1940   - vmcall
1941   0 .. 63
1942
1943 Additionally, the following IDs in 'vmx_exit' group are reserved and may
1944 not be used:
1945
1946   - Used by kernel: 1, 7, 41, 52, 55
1947
1948   - Reserved by Intel: 35, 38, 42
1949
1950 It is possible to assign event actions to event source entries.
1951 Currently supported source event actions are 'subject_sleep',
1952 'subject_yield', 'unmask_irq', 'system_reboot', 'system_poweroff' and
1953 'system_panic', which all have the kernel itself as endpoint.
1954 -->
1955 </event>
1956 <event id="1" logical="reset_linux" physical="reset_linux_1"/>
1957 <event id="2" logical="load_linux" physical="load_linux_1"/>
1958 </group>
1959 </source>
1960 <target>
1961 <!--
1962   The event 'target' element specifies events that the subject is an
1963   *endpoint* of.
1964 -->
1965 <event logical="resume_after_load" physical="start_linux_1">
1966 <!--
1967   The 'event' element in the target section specifies one event endpoint
1968   by referencing a physical event and assigning a logical name to it.
1969 -->
1970 </event>
1971 <event id="63" logical="reset" physical="reset_sm_1">
1972 <reset/>
1973 </event>
1974 </target>
1975 </events>
1976 <monitor>
1977 <!--
1978   The monitor abstraction enables subjects to request access to certain
1979   data of another subject specified by name. Possible child elements are:
1980
1981   - State
1982
1983   - Timed\_Events
1984
1985   - Interrupts
1986
1987   - Loader
1988
1989 See the Muen Component Specification document for details about these
1990 subject monitor interfaces.
1991 -->
1992 <state logical="monitor_state" subject="nic_linux" virtualAddress="16#001e_0000#" writable="true"/>
1993 <loader logical="reload" subject="nic_sm" virtualAddress="16#0000#">
1994 <!--
1995   The 'loader' mechanism effectively puts the loaded subject denoted by
1996   the 'subject' attribute under loader control, as it is not able to start
1997   without the help of the loader.
1998
1999 In more detail, the 'loader' monitor element instructs the expander tool
2000 to map all memory regions of the referenced subject into the address
2001 space of the monitor subject, using the specified 'virtualAddress' as
2002 offset in the address space of the loader.
2003
2004 If a memory region of the loaded subject is writable and file-backed,
2005 the region is replaced with an empty region and linked via the 'hashRef'
2006 mechanism to the original region which is mapped into the loader.
2007
2008 The state of the loaded subject is then invalidated by clearing the
2009 'CR4.VMXE' bit in the initial subject CR4 register value. If such a
2010 subject is scheduled by the kernel, a VMX exit +VM-entry failure due to
2011 invalid guest state* (33) occurs. See Intel SDM Vol. 3C, "23.7 Enabling
2012 and Entering VMX Operation" and Intel SDM Vol. 3C, "23.8 Restrictions on
2013 VMX Operation" for more details. This trap is linked to the loader via
2014 normal VMX event handling. After handover, the loader initializes the
2015 memory regions replaced by the expander with the designated content.

```

```

2017 All information required to *load* the loaded subject is provided to the
2018 loader subject via its own sinfo API. Memory regions prefixed with
2019 `monitor_sinfo_` provide access to the sinfo regions of the loaded
2020 subjects. Regions prefixed with `monitor_state_` specify memory regions
2021 containing the subject register state of the loaded subject.

2023 The difference between the `monitor_sinfo_` memory region address in the
2024 loader and the address of the `sinfo` memory region in the target sinfo
2025 information denotes the `virtualAddress` offset attribute of the
2026 `loader` element in the policy. This information combined is enough to
2027 fully construct the initial state of the loaded subject, or to reset a
2028 subject to its initial state on demand.

2029 The loader may also optionally check the hashes of the restored regions,
2030 as this information is provided via the sinfo mechanism as well.
-->
2033 </loader>
2034 </monitor>
2035 <component ref="sm">
2036   <map logical="time_info" physical="time_info"/>
2037   <map logical="debuglog" physical="debuglog_subject2"/>
2038   <map logical="dm_pciconf_req" physical="nic_dm_request"/>
2039   <map logical="dm_pciconf_res" physical="nic_dm_response"/>
2040   <map logical="resume_subject" physical="resume_linux_1"/>
2041   <map logical="handle_subject_trap" physical="trap_to_sm_1"/>
2042   <map logical="status" physical="status_sm_1"/>
2043   <map logical="control" physical="control_sm_1"/>
2044 </component>
2045 </subject>
2046 <subject name="storage_sm">
2047   <events>
2048     <source>
2049       <group name="vmcall">
2050         <event id="0" logical="serial_irq4" physical="serial_irq4_linux_2"/>
2051         <event id="1" logical="reset_linux" physical="reset_linux_2"/>
2052       </group>
2053       <group name="vmx_exit">
2054         <default physical="system_panic">
2055           <system_panic/>
2056         </default>
2057       </group>
2058     </source>
2059   </events>
2060 </monitor>
2061 <state logical="monitor_state" subject="storage_linux" virtualAddress="16#001e_0000#" writable="true
2062   "/>
2063   <loader logical="reload" subject="storage_sm" virtualAddress="16#0000#"/>
2064 </monitor>
2065 <component ref="sm">
2066   <map logical="time_info" physical="time_info"/>
2067   <map logical="debuglog" physical="debuglog_subject3"/>
2068   <map logical="dm_pciconf_req" physical="storage_dm_request"/>
2069   <map logical="dm_pciconf_res" physical="storage_dm_response"/>
2070   <map logical="resume_subject" physical="resume_linux_2"/>
2071   <map logical="handle_subject_trap" physical="trap_to_sm_2"/>
2072   <map logical="status" physical="status_sm_2"/>
2073   <map logical="control" physical="control_sm_2"/>
2074 </component>
2075 </subject>
2076 <subject name="time">
2077   <events>
2078     <source>
2079       <group name="vmx_exit">
2080         <default physical="system_panic">
2081           <system_panic/>
2082         </default>
2083       </group>
2084     </source>
2085   </events>
2086 <component ref="time">
2087   <map logical="time_export1" physical="time_info"/>
2088   <map logical="debuglog" physical="debuglog_subject4"/>
2089   <map logical="cmos_rtc" physical="cmos_rtc">
2090     <map logical="ports" physical="ports"/>
2091   </map>
2092   <map logical="status" physical="status_time"/>
2093   <map logical="control" physical="control_time"/>
2094 </component>
2095 </subject>
2096 <subject name="nic_sl">
2097   <events>
2098     <source>
2099       <group name="vmx_exit">
2100         <default physical="system_panic">
2101           <system_panic/>
2102         </default>

```

```

2103 </group>
2104 </source>
2105 <events>
2106 <monitor>
2107   <loader logical="monitor_loader_nic_linux" subject="nic_linux" virtualAddress="16#0001_0000_0000#" />
2108 </monitor>
2109   <component ref="sl">
2110     <map logical="start" physical="start_linux_1"/>
2111     <map logical="handle_reset" physical="load_linux_1"/>
2112     <map logical="status" physical="status_linux_1"/>
2113     <map logical="control" physical="control_linux_1"/>
2114   </component>
2115 </subject>
2116 <subject name="ps2">
2117   <events>
2118     <source>
2119       <group name="vmx_exit">
2120         <default physical="system_panic">
2121           <system_panic/>
2122         </default>
2123       </group>
2124     </source>
2125   </events>
2126   <component ref="ps2_drv">
2127     <map logical="input_events" physical="input_events"/>
2128     <map logical="debuglog" physical="debuglog_subject5"/>
2129     <map logical="ps2" physical="ps2">
2130       <map logical="kbd_irq" physical="kbd_irq"/>
2131       <map logical="mouse_irq" physical="mouse_irq"/>
2132       <map logical="port_60" physical="port_60"/>
2133       <map logical="port_64" physical="port_64"/>
2134     </map>
2135   </component>
2136 </subject>
2137 <subject name="example">
2138   <events>
2139     <source>
2140       <group name="vmx_exit">
2141         <default physical="system_panic">
2142           <system_panic/>
2143         </default>
2144       </group>
2145     </source>
2146   </events>
2147   <monitor>
2148     <state logical="monitor_state" subject="storage_linux" virtualAddress="16#001e_0000#" writable="true"
2149       "/>
2150     <loader logical="reload" subject="example" virtualAddress="16#0000#"/>
2151   </monitor>
2152   <component ref="example">
2153     <map logical="example_request" physical="example_request"/>
2154     <map logical="example_response" physical="example_response"/>
2155     <map logical="debuglog" physical="debuglog_example"/>
2156     <map logical="sleep" physical="subject_sleep"/>
2157     <map logical="yield" physical="subject_yield"/>
2158     <map logical="timer" physical="example_self"/>
2159     <map logical="inject_timer" physical="example_self"/>
2160     <map logical="control" physical="control_example"/>
2161     <map logical="status" physical="status_example"/>
2162   </component>
2163 </subject>
2164 <subject name="controller">
2165   <events>
2166     <source>
2167       <group name="vmx_exit">
2168         <default physical="system_panic">
2169           <system_panic/>
2170         </default>
2171       </group>
2172     </source>
2173   </events>
2174   <component ref="controller">
2175     <map logical="debuglog" physical="debuglog_controller"/>
2176     <map logical="control_1" physical="control_time"/>
2177     <map logical="control_2" physical="control_sm_1"/>
2178     <map logical="control_3" physical="control_sm_2"/>
2179     <map logical="control_4" physical="control_example"/>
2180     <map logical="control_5" physical="control_linux_1"/>
2181     <map logical="status_1" physical="status_time"/>
2182     <map logical="status_2" physical="status_sm_1"/>
2183     <map logical="status_3" physical="status_sm_2"/>
2184     <map logical="status_4" physical="status_example"/>
2185     <map logical="status_5" physical="status_linux_1"/>
2186     <map logical="slot_control_1" physical="slot_control_1"/>
2187     <map logical="reset_slot_1_sm" physical="reset_sm_1"/>
2188     <map logical="reset_slot_1_linux" physical="reset_slot_1"/>
2189   </component>

```

```

2189 </subject>
2190 <subject name="nic_dm">
2191 <devices>
2192 <!--
2193 List of device references. Used to grant a subject access to hardware
2194 devices and their resources.
2195 -->
2196 <device logical="nic" physical="ethernet_controller_1">
2197 <!--
2198 The 'device' element allows a subject access to devices referenced via
2199 the 'physical' attribute.
2200
2201 For PCI devices only a single virtual bus is provided (bus 0). The 'pci'
2202 element may be used to place the device at a specific location (BDF). If
2203 no other logical device resources of the device are specified, then the
2204 expander tool will map all physical devices resources into the subject.
2205 When logical device resources are explicitly specified, then only access
2206 to those are actually granted. The physical attribute must be either a
2207 reference to an existing physical device, device alias or device class.
2208 Validators check that this is the case.
2209 -->
2210 <pci bus="16#00#" device="16#01#" function="0"/>
2211 <memory executable="false" logical="mmconf" physical="mmconf" writable="true">
2212 <!--
2213 The device 'memory' element maps the device memory region referenced via
2214 the 'physical' attribute into the subject address space at address
2215 'virtualAddress'. The 'executable', 'writable' attributes define the
2216 access permissions for the subject.
2217 -->
2218 </memory>
2219 </device>
2220 </devices>
2221 <events>
2222 <source>
2223 <group name="vmx_exit">
2224 <default physical="system_panic">
2225 <system_panic/>
2226 </default>
2227 </group>
2228 </source>
2229 </events>
2230 <component ref="dm">
2231 <map logical="debuglog" physical="debuglog_subject6"/>
2232 <map logical="request" physical="nic_dm_request"/>
2233 <map logical="response" physical="nic_dm_response"/>
2234 </component>
2235 </subject>
2236 <subject name="storage_dm">
2237 <events>
2238 <source>
2239 <group name="vmx_exit">
2240 <default physical="system_panic">
2241 <system_panic/>
2242 </default>
2243 </group>
2244 </source>
2245 </events>
2246 <component ref="dm">
2247 <map logical="debuglog" physical="debuglog_subject7"/>
2248 <map logical="request" physical="storage_dm_request"/>
2249 <map logical="response" physical="storage_dm_response"/>
2250 </component>
2251 </subject>
2252 <subject name="dbgserver">
2253 <events>
2254 <source>
2255 <group name="vmx_exit">
2256 <default physical="system_panic">
2257 <system_panic/>
2258 </default>
2259 </group>
2260 </source>
2261 </events>
2262 <component ref="dbgserver">
2263 <map logical="log_channel1" physical="debuglog_subject1"/>
2264 <map logical="log_channel2" physical="debuglog_subject2"/>
2265 <map logical="log_channel3" physical="debuglog_subject3"/>
2266 <map logical="log_channel4" physical="debuglog_subject4"/>
2267 <map logical="log_channel5" physical="debuglog_subject5"/>
2268 <map logical="log_channel_example" physical="debuglog_example"/>
2269 <map logical="log_channel_6" physical="debuglog_controller"/>
2270 <map logical="log_channel7" physical="debuglog_subject6"/>
2271 <map logical="log_channel8" physical="debuglog_subject7"/>
2272 <map logical="crash_audit" physical="crash_audit"/>
2273 <map logical="debugconsole" physical="serial_device_1">
2274 <map logical="port" physical="ioport1"/>
2275 </map>
```

```

2275 <map logical="reboot" physical="system_reboot"/>
2276 <map logical="shutdown" physical="system_poweroff"/>
2277 </component>
2278 </subject>
2279 <subject name="nic_linux">
2280 <bootparams>console=hvc console=ttyS0 hostname=nic_linux</bootparams>
2281 <memory>
2282 <memory executable="false" logical="initramfs" physical="initramfs" virtualAddress="16#7000_0000#" writable="false"/>
2283 </memory>
2284 <devices>
2285 <device logical="eth0" physical="nic_1">
2286 <pci bus="16#00#" device="16#01#" function="0"/>
2287 </device>
2288 <device logical="additional_nics" physical="additional_nics"/>
2289 </devices>
2290 <events>
2291 <source>
2292 <group name="vmx_exit">
2293 <default physical="trap_to_sm_1"/>
2294 </group>
2295 <group name="vmcall">
2296 <event id="30" logical="reboot" physical="reboot_linux_1"/>
2297 <event id="31" logical="timer" physical="timer_linux_1"/>
2298 </group>
2299 </source>
2300 <target>
2301 <event logical="resume_after_trap" physical="resume_linux_1"/>
2302 <event id="63" logical="reset" physical="reset_linux_1">
2303 <reset/>
2304 </event>
2305 <event id="62" logical="reset_from_vt" physical="reset_slot_1">
2306 <reset/>
2307 </event>
2308 <event id="61" logical="reboot" physical="reboot_linux_1">
2309 <reset/>
2310 </event>
2311 <event logical="serial_irq4" physical="serial_irq4_linux_1">
2312 <inject_interrupt vector="52">
2313 <!--
2314   Instructs the SK to inject a guest interrupt with given vector on event
2315   occurrence.
2316   -->
2317 </inject_interrupt>
2318 </event>
2319 <event logical="timer" physical="timer_linux_1">
2320 <inject_interrupt vector="236"/>
2321 </event>
2322 </target>
2323 </events>
2324 <channels>
2325 <!--
2326   The 'channel' section of a subject declares references to communication
2327   channels. The referenced channels become accessible to the requesting
2328   subject either as reader or writer endpoint.
2329   -->
2330 <reader logical="virtual_input" physical="virtual_input_1" vector="100" virtualAddress="16#3000#">
2331 <!--
2332   A channel 'reader' element references a global communication channel as
2333   reader endpoint, i.e. the channel is mapped read-only into the subject
2334   address space.
2335   -->
2336 </reader>
2337 <writer event="1" logical="virtual_console" physical="virtual_console_1" virtualAddress="16#4000#">
2338 <!--
2339   A channel 'writer' element references a global communication channel as
2340   writer endpoint, i.e. the channel is mapped with write permissions into
2341   the subject address space.
2342   -->
2343 </writer>
2344 <reader logical="testchannel_2" physical="testchannel_2" virtualAddress="16#000e_00f0_0000#" />
2345 <writer logical="testchannel_1" physical="testchannel_1" virtualAddress="16#000e_00f0_1000#" />
2346 <reader logical="testchannel_4" physical="testchannel_4" virtualAddress="16#000e_00f0_2000#" />
2347 <writer logical="testchannel_3" physical="testchannel_3" virtualAddress="16#000e_0100_2000#" />
2348 </channels>
2349 <component ref="linux">
2350 <map logical="lowmem" physical="nic_linux|lowmem"/>
2351 <map logical="ram" physical="nic_linux|ram"/>
2352 </component>
2353 </subject>
2354 <subject name="storage_linux">
2355 <bootparams>console=hvc console=ttyS0 hostname=storage_linux</bootparams>
2356 <memory>
2357 <memory executable="false" logical="initramfs" physical="initramfs" virtualAddress="16#7000_0000#" writable="false"/>
2358 </memory>
2359 <devices>

```

```

2361 <device logical="xhci" physical="usb_controller_1"/>
2362 </devices>
2363 <events>
2364 <source>
2365 <group name="vmx_exit">
2366 <default physical="trap_to_sm_2"/>
2367 </group>
2368 <group name="vmcall">
2369 <event id="31" logical="timer" physical="timer_linux_2"/>
2370 </group>
2371 </source>
2372 <target>
2373 <event logical="resume_after_trap" physical="resume_linux_2"/>
2374 <event id="63" logical="reset" physical="reset_linux_2">
2375 <reset/>
2376 </event>
2377 <event logical="serial_irq4" physical="serial_irq4_linux_2">
2378 <inject_interrupt vector="52"/>
2379 </event>
2380 <event logical="timer" physical="timer_linux_2">
2381 <inject_interrupt vector="236"/>
2382 </event>
2383 </target>
2384 </events>
2385 <channels>
2386 <reader logical="virtual_input" physical="virtual_input_2" vector="49" virtualAddress="16#3000#"/>
2387 <writer event="1" logical="virtual_console" physical="virtual_console_2" virtualAddress="16#4000#"/>
2388 <reader logical="example_response" physical="example_response" vector="102" virtualAddress="16#000
2389   e_0210_0000#"/>
2390 <writer event="2" logical="example_request" physical="example_request" virtualAddress="16#000
2391   e_0210_1000#"/>
2392 <reader logical="testchannel_1" physical="testchannel_1" virtualAddress="16#000e_00f0_0000#"/>
2393 <writer logical="testchannel_2" physical="testchannel_2" virtualAddress="16#000e_00f0_1000#"/>
2394 <reader logical="testchannel_3" physical="testchannel_3" virtualAddress="16#000e_00f0_2000#"/>
2395 <writer logical="testchannel_4" physical="testchannel_4" virtualAddress="16#000e_0100_2000#"/>
2396 </channels>
2397 <component ref="linux">
2398 <map logical="lowmem" physical="storage_linux|lowmem"/>
2399 <map logical="ram" physical="storage_linux|ram"/>
2400 </component>
2401 </subject>
2402 </subjects>
2403 <scheduling tickRate="100000">
2404 <!--
2405 The Muen SK implements a fixed, cyclic scheduler. The 'scheduling'
2406 element is used to specify such a static plan by means of a major frame.
2407 A major frame consist of an arbitrary number of minor frames. Minor
2408 frames in turn specify a duration in number of ticks a scheduling
2409 partition is scheduled.
2410
2411 Scheduling partitions defined in the 'partitions' element consist of one
2412 or more scheduling groups, which in turn specify one or more subjects to
2413 be scheduled. *Scheduling groups* are used to define groups of
2414 cooperating subjects, which are allowed to hand over execution to a
2415 subject in the same scheduling group. This is done via *handover*
2416 events. Membership of a scheduling group must be specified explicitly in
2417 the policy, validators enforce that these settings are correct by
2418 calculating the chain of handover events.
2419
2420 While scheduling groups support the efficient cooperation of multiple
2421 subjects, subjects which need to be spatially but not temporally
2422 isolated from each other cannot profit from it. To efficiently support
2423 this use-case, the scheduling partition concept is implemented.
2424
2425 Within a *scheduling partition*, all scheduling groups are scheduled
2426 round robin with preemption and the opportunity to yield and/or sleep.
2427 If a subject in a scheduling group sleeps or yields, the next scheduling
2428 group in the scheduling partition is scheduled. More precisely: the
2429 active subject of the next scheduling group is executed by the SK.
2430
2431 Note that prioritization is not implemented on purpose to avoid any
2432 starvation issues\[1\]. The yield operation maps to the x86\_64 'PAUSE'
2433 instruction, while sleep corresponds to 'HLT'. See the *Muen Component
2434 Specification* document for more information on this topic.
2435
2436 Minor frames designate the scheduling partition that is to be executed
2437 for the given amount of ticks. The scheduling partition attribute 'name'
2438 uniquely identifies a scheduling partition. On first activation, the
2439 first scheduling group (in XML-order) is scheduled. Within the
2440 scheduling group, the first subject (again in XML-order) is executed.
2441 The active subject of a scheduling group may change over time, as the
2442 cooperating subjects initiate handover events.
2443
2444 The tickRate attribute of the 'scheduling' element has the unit Hertz
2445 (Hz) and specifies the number of clock ticks per second. The ticks
2446 attribute of minor frames is expressed in terms of this tick rate. As an
2447 example: if we want to declare the minor frame duration in terms of

```

```

2445 microseconds (\(10^{-6}\))) then a tick rate of 1000000 must be used.
2447 The duration of a major frame must be the same on each CPU, meaning the
2448 sum of all minor frame ticks for any given CPU must be identical.
2449 However, different major frames can have arbitrary length.
2451 The Tau0 subject designates to the kernel which major frame is the
2452 currently active one. At the end of each major frame, the kernel
2453 determines the active major frame and switches to that scheduling plan
2454 for the duration of the major frame.
2455 1. Prioritization with starvation protection cannot be implemented with
2456 low complexity
-->
2459 <partitions>
2460   <partition name="nic_linux">
2461     <group>
2462       <subject name="nic_linux"/>
2463       <subject name="nic_sm"/>
2464       <subject name="nic_sl"/>
2465       <subject name="nic_dm"/>
2466     </group>
2467   </partition>
2468   <partition name="ps2_driver">
2469     <group>
2470       <subject name="ps2"/>
2471     </group>
2472   </partition>
2473   <partition name="controller">
2474     <group>
2475       <subject name="controller"/>
2476     </group>
2477   </partition>
2478   <partition name="idle_0">
2479     <group>
2480       <subject name="mugenschedcfg_auto_idle_0"/>
2481     </group>
2482   </partition>
2483   <partition name="storage_linux">
2484     <group>
2485       <subject name="storage_linux"/>
2486       <subject name="storage_sm"/>
2487       <subject name="storage_dm"/>
2488     </group>
2489   </partition>
2490   <subject name="example"/>
2491 </group>
2492 </partition>
2493 <partition name="debugserver">
2494   <group>
2495     <subject name="dbgserver"/>
2496   </group>
2497 </partition>
2498 <partition name="time">
2499   <group>
2500     <subject name="time"/>
2501   </group>
2502 </partition>
2503 <partition name="vt">
2504   <group>
2505     <subject name="vt"/>
2506   </group>
2507 </partition>
2508 <partition name="tau0">
2509   <group>
2510     <subject name="tau0"/>
2511   </group>
2512 </partition>
2513 <partition name="idle_1">
2514   <group>
2515     <subject name="mugenschedcfg_auto_idle_1"/>
2516   </group>
2517 </partition>
</partitions>
<majorFrame>
<!--
2521 A major frame consists of a sequence of minor frames for a given CPU.
2522 When the end of a major frame is reached, all CPUs synchronize and the
2523 scheduler starts over from the beginning using the first minor frame
2524 again. This means that major frames are repeated in a cyclic fashion
2525 until a different major frame is designated via the Tau0 interface.
-->
2527 <cpu id="0">
<!--
2529 The 'cpu' element is used to specify major frames for each CPU of the
2530 system.
-->

```



```

2793 <minorFrame partition="idle_0" ticks="1"/>
2795 <minorFrame partition="nic_linux" ticks="4"/>
2797 <minorFrame partition="idle_0" ticks="1"/>
2799 <minorFrame partition="nic_linux" ticks="4"/>
2801 <minorFrame partition="idle_0" ticks="1"/>
2803 <minorFrame partition="nic_linux" ticks="4"/>
2805 <minorFrame partition="idle_0" ticks="1"/>
2807 <minorFrame partition="nic_linux" ticks="4"/>
2809 <minorFrame partition="idle_0" ticks="1"/>
2811 <minorFrame partition="nic_linux" ticks="4"/>
2813 <minorFrame partition="idle_0" ticks="1"/>
2815 <minorFrame partition="nic_linux" ticks="4"/>
2817 <minorFrame partition="idle_0" ticks="1"/>
2819 <minorFrame partition="nic_linux" ticks="4"/>
2821 <minorFrame partition="idle_0" ticks="1"/>
2823 <minorFrame partition="nic_linux" ticks="4"/>
2825 <minorFrame partition="idle_0" ticks="1"/>
2827 <minorFrame partition="nic_linux" ticks="4"/>
2829 <minorFrame partition="idle_0" ticks="1"/>
2831 <minorFrame partition="nic_linux" ticks="4"/>
2833 <minorFrame partition="idle_0" ticks="1"/>
2835 <cpu id="1">
2837 <minorFrame partition="storage_linux" ticks="3"/>
2839 <minorFrame partition="debugserver" ticks="2"/>
2841 <minorFrame partition="storage_linux" ticks="3"/>
2843 <minorFrame partition="debugserver" ticks="1"/>
2845 <minorFrame partition="time" ticks="1"/>
2847 <minorFrame partition="storage_linux" ticks="3"/>
2849 <minorFrame partition="storage_linux" ticks="2"/>
2851 <minorFrame partition="idle_1" ticks="2"/>
2853 <minorFrame partition="storage_linux" ticks="3"/>
2855 <minorFrame partition="debugserver" ticks="2"/>
2857 <minorFrame partition="storage_linux" ticks="3"/>
2859 <minorFrame partition="idle_1" ticks="1"/>
2861 <minorFrame partition="storage_linux" ticks="2"/>
2863 <minorFrame partition="idle_1" ticks="3"/>
2865 <minorFrame partition="storage_linux" ticks="2"/>
2867 <minorFrame partition="idle_1" ticks="2"/>
2869 <minorFrame partition="storage_linux" ticks="3"/>
2871 <minorFrame partition="debugserver" ticks="2"/>
2873 <minorFrame partition="storage_linux" ticks="1"/>
2875 <minorFrame partition="idle_1" ticks="3"/>
2877 <minorFrame partition="storage_linux" ticks="2"/>
2879 <minorFrame partition="idle_1" ticks="3"/>

```

```

2881 <minorFrame partition="idle_1" ticks="2"/>
2882 <minorFrame partition="storage_linux" ticks="3"/>
2883 <minorFrame partition="idle_1" ticks="2"/>
2884 <minorFrame partition="storage_linux" ticks="3"/>
2885 <minorFrame partition="idle_1" ticks="2"/>
2886 <minorFrame partition="storage_linux" ticks="3"/>
2887 <minorFrame partition="debugserver" ticks="2"/>
2888 <minorFrame partition="storage_linux" ticks="3"/>
2889 <minorFrame partition="debugserver" ticks="1"/>
2890 <minorFrame partition="idle_1" ticks="1"/>
2891 <minorFrame partition="storage_linux" ticks="3"/>
2892 <minorFrame partition="vt" ticks="2"/>
2893 <minorFrame partition="storage_linux" ticks="3"/>
2894 <minorFrame partition="storage_linux" ticks="2"/>
2895 <minorFrame partition="storage_linux" ticks="3"/>
2896 <minorFrame partition="idle_1" ticks="2"/>
2897 <minorFrame partition="storage_linux" ticks="3"/>
2898 <minorFrame partition="idle_1" ticks="2"/>
2899 <minorFrame partition="storage_linux" ticks="3"/>
2900 <minorFrame partition="idle_1" ticks="2"/>
2901 <minorFrame partition="storage_linux" ticks="3"/>
2902 <minorFrame partition="idle_1" ticks="2"/>
2903 <minorFrame partition="storage_linux" ticks="3"/>
2904 <minorFrame partition="debugserver" ticks="2"/>
2905 <minorFrame partition="storage_linux" ticks="3"/>
2906 <minorFrame partition="debugserver" ticks="1"/>
2907 <minorFrame partition="idle_1" ticks="1"/>
2908 <minorFrame partition="storage_linux" ticks="3"/>
2909 <minorFrame partition="vt" ticks="2"/>
2910 <minorFrame partition="storage_linux" ticks="3"/>
2911 <minorFrame partition="storage_linux" ticks="2"/>
2912 <minorFrame partition="idle_1" ticks="3"/>
2913 <minorFrame partition="storage_linux" ticks="3"/>
2914 <minorFrame partition="idle_1" ticks="2"/>
2915 <minorFrame partition="storage_linux" ticks="3"/>
2916 <minorFrame partition="idle_1" ticks="2"/>
2917 <minorFrame partition="storage_linux" ticks="3"/>
2918 <minorFrame partition="idle_1" ticks="2"/>
2919 <minorFrame partition="storage_linux" ticks="3"/>
2920 <minorFrame partition="debugserver" ticks="2"/>
2921 <minorFrame partition="storage_linux" ticks="3"/>
2922 <minorFrame partition="debugserver" ticks="1"/>
2923 <minorFrame partition="idle_1" ticks="1"/>
2924 <minorFrame partition="storage_linux" ticks="3"/>
2925 <minorFrame partition="vt" ticks="2"/>
2926 <minorFrame partition="storage_linux" ticks="3"/>
2927 <minorFrame partition="storage_linux" ticks="2"/>
2928 <minorFrame partition="storage_linux" ticks="3"/>
2929 <minorFrame partition="idle_1" ticks="2"/>
2930 <minorFrame partition="storage_linux" ticks="3"/>
2931 <minorFrame partition="idle_1" ticks="2"/>
2932 <minorFrame partition="storage_linux" ticks="3"/>
2933 <minorFrame partition="idle_1" ticks="2"/>
2934 <minorFrame partition="storage_linux" ticks="3"/>
2935 <minorFrame partition="idle_1" ticks="2"/>
</cpu>
2936 </majorFrame>
2937 </scheduling>
2938 </system>

```

Listing 8.1: Demo System (VT-d)

Chapter 9

Bibliography

- [1] Adrian-Ken Rueeggsegger and Reto Buerki. *Muen Component Specification*. 2021.
- [2] Adrian-Ken Rueeggsegger and Reto Buerki. *Muen Separation Kernel*. 2021.