Muen Component Specification

Adrian-Ken Rueegsegger, Reto Buerki

v0.7.2, April 9, 2024

無縁

Contents

List of Figures

Listings

Chapter 1

Introduction

In a component-based software architecture (*CBA*), system functionality is realized by small, unprivileged applications, so called *components*.

The Muen Separation Kernel (SK) is a specialized microkernel, which allows the controlled execution of a software component as a *subject*. Communication among subjects, and also between a subject and the kernel itself is strictly regulated by a system policy, which is enforced by the Muen SK.

This document defines the foundation of Muen components, the interfaces, parameterization, documentation and verification. To illustrate these topics, an actual, runnable *example* component is used.

This document has been generated by analyzing the annotated source code of the example component and related sources, using git repository revision `e878fd944581`.

The following subsections first define the terms *component*, *subject* and *library*, before the actual component mechanisms are explained.

1.1 Components

A component is a piece of software to be executed by the Muen separation kernel within a well-defined execution environment. A Muen component consists of the following parts:

- Source code which is compiled into an executable binary
- Component specification
- Documentation

The component specification 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

Listing 1.1 shows the component specification XML of the example component. First, the component defines configuration options which are used to parameterize the component (section 4), then software library dependencies are declared (section 1.3). The `requires` section specifies the *expectations* the component has for its execution environment.

```xml
<?xml version="1.0" encoding="utf-8"?>
<component name="example" profile="native">
 <config>
  <boolean name="ahci_drv_enabled" value="false"/>
  <boolean name="print_serial" value="false"/>
  <boolean name="print_vcpu_speed" value="true"/>
  <integer name="serial" value="123456789"/>
  <string name="greeter" value="Subject running"/>
 </config>
 <depends>
  <library ref="libmudebuglog"/>
  <library ref="muinit"/>
 </depends>
 <requires>
  <vcpu>
   <vmx>
    <masks>
     <exception>
      <Breakpoint>0</Breakpoint>
     </exception>
    </masks>
   </vmx>
   <registers>
    <gpr>
     <rip>16#0020_0000#</rip>
    </gpr>
    <cr4>
     <XSAVEEnable>1</XSAVEEnable>
    </cr4>
   </registers>
  </vcpu>
  <memory>
   <memory executable="false" logical="filled_region" size="16#1000#" virtualAddress="
    16#0001_0000_0000#" writable="true"/>
  </memory>
  <channels>
   <reader logical="example_request" size="16#1000#" vector="64" virtualAddress="16#0001
    _0000_1000#"/>
   <writer event="16" logical="example_response" size="16#1000#" virtualAddress="16#0001
    _0000_2000#"/>
  </channels>
  <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>
 </requires>
 <provides>
  <memory executable="false" logical="interrupt_stack" size="16#2000#" type="
    subject_binary" virtualAddress="16#0001_0000#" writable="true">
   <fill pattern="16#00#"/>
  </memory>
  <memory executable="true" logical="text" size="16#4000#" type="subject_binary"
    virtualAddress="16#0020_0000#" writable="false">
   <file filename="example_text" offset="none"/>
   <hash value="16#f09a98fdd53015ba2c2484b330b68bbad129d60054a6f610f26e9efe300fb379#"/>
  </memory>
  <memory executable="false" logical="rodata" size="16#1000#" type="subject_binary"
    virtualAddress="16#0020_4000#" writable="false">
   <file filename="example_rodata" offset="none"/>
   <hash value="16#c647749cba2a151be2a1c451441bfc4882e76f8a554c74ee497dfdcc55b70785#"/>
  </memory>
  <memory executable="false" logical="data" size="16#1000#" type="subject_binary"
    virtualAddress="16#0020_5000#" writable="true">
   <file filename="example_data" offset="none"/>
   <hash value="16#3466ddf188d8d88cef240e0f02dedb3c09d5a21d6c27b3f3299b74dcd3e30393#"/>
  </memory>
  <memory executable="false" logical="bss" size="16#3000#" type="subject_binary"
    virtualAddress="16#0020_6000#" writable="true">
   <fill pattern="16#00#"/>
  </memory>
  <memory executable="false" logical="stack" size="16#2000#" type="subject_binary"
    virtualAddress="16#1000#" writable="true">
   <fill pattern="16#00#"/>
  </memory>
 </provides>
</component>
```

Listing 1.1: Component Specification XML

The example component expects the following shared memory channels:

- `example_request`
- `example_response`

These are used to retrieve requests from a client, and to send a response (section 9).

The example component also expects the availability of three source events with given IDs, and it is the target endpoint for the `inject_timer` event.

An integrator must link the logical resources of a component specification with the actual system policy. How this is done is explained in section 1.2.

The `provides` section specifies memory regions which are provided by the component. Such regions will be added to the physical memory regions of the system and linked into the subject referencing the component by the `mucfgexpand` tool. Usually, the provided regions consist of the component binary, but it is possible to provide arbitrary regions, with or without content.

In order to simplify the integrators job, the `mucbinsplit` tool can be used to generate the component binary regions with the correct permissions automatically. See the *Muen System Specification* [1] document for a detailed description of the tool.

1.2 Subjects

A subject is an instance of a component, i.e. an active entity that is executed by the Muen kernel during runtime. Its specification references a component and maps the requested logical resources to physical resources provided by the system.

```
<subject name="example">
 <events>
  <source>
   <group name="vmx_exit">
    <default physical="system_panic">
     <system_panic/>
    </default>
   </group>
  </source>
 </events>
 <monitor>
  <state subject="storage_linux" logical="monitor_state" virtualAddress="16#001e_0000#"
    writable="false"/>
  <loader logical="reload" subject="example" virtualAddress="16#0000#"/>
 </monitor>
 <component ref="example">
  <map logical="example_request" physical="example_request"/>
  <map logical="example_response" physical="example_response"/>
  <map logical="debuglog" physical="debuglog_example"/>
  <map logical="sleep" physical="subject_sleep"/>
  <map logical="yield" physical="subject_yield"/>
  <map logical="timer" physical="example_self"/>
  <map logical="inject_timer" physical="example_self"/>
  <map logical="control" physical="control_example"/>
  <map logical="status" physical="status_example"/>
  <if variable="ahci_drv_active" value="true">
   <map logical="blockdev_request2" physical="blockdev_request2"/>
   <map logical="blockdev_response2" physical="blockdev_response2"/>
   <map logical="blockdev_shm2" physical="blockdev_shm2"/>
  </if>
  <map logical="filled_region" physical="example_filled_region"/>
 </component>
</subject>
```

Listing 1.2: Mapping of component resources

In listing 1.2, the logical channels `example_request`, `example_reponse` and `debuglog` of the example component are mapped to physical channels of the system policy via `map` elements. The `debuglog` channel is not directly visible in the 1.2 listing, as it is added by the dependency to the libmudebuglog library (which is declared in the component XML listing `depends` section).

Also, the requested source and target events are mapped to global system events.

Validators enforce that all requested resources of a component are properly mapped by the subject.

Note that subjects do not need to have all resources specified in their component description. Instead, extra resources such as device and/or memory mappings can be assigned to a subject. This is useful for components which are able to discover the available resources at runtime by using configuration mechanism like PCI configuration space enumeration, ACPI or the Muen subject information page API specified in section 9.4.

1.3 Libraries

Code which is used by multiple software components is usually factored out into a library to reduce code duplication. This is also possible for Muen components. In contrast to shared dynamic libraries of a generic multi-purpose OS, only the source code and policy resources are shared, not the address space of the library itself. The code lives separately in all components which use the library. Therefore, Muen component libraries can be seen as a static library in generic OS terms.

A Muen *component library* provides a library specification, like the example provided in listing 1.3. Muen components declare a dependency on a library in the XML specification as outlined in section 1.1. If the library in question is written in SPARK/Ada, the component project also declares the dependency in its GNAT project file. After that, the functionality provided by the software library is accessible from the component source code.

```
<library name="libmudebuglog">
  <include href="config.xml"/>
  <requires>
    <channels>
      <writer logical="debuglog" virtualAddress="16#000f_fff0_0000#" size="$
      logchannel_size"/>
    </channels>
  </requires>
</library>
```

Listing 1.3: Libmudebuglog library specification XML

Similar to components, the library `requires` certain resources to operate. In this case, a `debuglog` channel is expected at the given `virtualAddress` with the specified `size`. Note that the size is provided by a variable. The library intends to write to this shared memory channel as a channel `writer`. A component depending on this library inherits the requested resource from the library, and the system integrator must map them to physical system resources at the subject level.

The example XML specification also contains an `include` directive for another XML file: `config.xml`. This mechanism is used to generate configuration settings during the build depending on the actual system policy. This task is component-specific and can be implemented via various solutions, e.g. an XSLT transform or a Python script to name just two. In this case, the size of the debugserver log channel is extracted from the system policy, see listing 1.4. More specifically, it contains the value for the `$logchannel_size` variable mentioned before in listing 1.3.

```
<config>
 <string name="logchannel_size" value="16#0002_0000#"/></config>
```

Listing 1.4: Libmudebuglog generated config.xml

Chapter 2

Resource Discovery

In order to interact with the environment, components must know the properties of resources like channels, memory or hardware devices. Depending on the actual resource, possible resource properties are virtual address, memory size, event or I/O port numbers.

The Muen platform provides two main methods to learn about such properties. The main difference is the point in time the value of a property is acquired:

- Build-time static
- Dynamic discovery at runtime

While the first mechanism is mainly used by native (SPARK/Ada) components, the second mechanism is often applied by a traditional guest virtual machine with a full-fledged OS. Since all resources are already known at compile-time, and because native (verified) components should be kept as simple as possible, they refrain from dynamic discovery and use source code constants generated by the CSPECS mechanism described in section 2.1.

An operating system like Linux tends to discover its resources at runtime. This is possible via the sinfo API introduced in section 2.2 and specified in detail in section 9.4.

While resources can be discovered via the sinfo and Component Specifications (CSPECS) mechanisms, there exist OS-specific methods to determine available resources, which are also supported by the Muen platform. They are described in section 2.3.

It is also possible to combine multiple methods as needed, like it is done in the example component or in the Muen Linux guest operating system where both generated ACPI tables and the sinfo API are employed to discover resources.

2.1 CSPECS Mechanism

During the component build process, the component specification in XML format outlined in section 1.1 and 1.3 is translated to SPARK/Ada specifications using the `mucgenspec` tool, see [1].

The generated packages in the `Example_Component` hierarchy contain resource properties as constants for SPARK/Ada, see listing 2.1 as an example. It contains all the constants required to access channel resources.

```
pragma Style_Checks (Off);

package Example_Component.Channels
is

   Example_Request_Address : constant := 16#0001_0000_1000#;
   Example_Request_Size    : constant := 16#1000#;
   Example_Request_Kind    : constant Channel_Kind := Channel_Reader;
   Example_Request_Vector  : constant := 64;

   Example_Response_Address : constant := 16#0001_0000_2000#;
   Example_Response_Size    : constant := 16#1000#;
   Example_Response_Kind    : constant Channel_Kind := Channel_Writer;
   Example_Response_Event   : constant := 16;
```

```
end Example_Component.Channels;
```

Listing 2.1: CSPECS constants for SPARK/Ada

The constants can be used in the source code of a component as illustrated in listing 2.2.

```
   Request : Foo.Message_Type
     with
       Volatile,
       Async_Writers,
       Address => System'To_Address
         (Example_Component.Channels.Example_Request_Address);
```

Listing 2.2: Usage of CSPECS constants

The Ada record type representing the example request channel data is placed at the memory location designated in the component XML specification using Ada `Address` and `Volatile` aspects in combination with the CSPECs generated resource constants.

2.2 Subject Information (sinfo) Mechanism

The Muen subject information (*sinfo*) and scheduling information (*schedinfo*) APIs allow the discovery of resource and scheduling data at runtime.

The information is provided by mapping the subject info and scheduling info pages into the virtual address space of a subject. This task is performed automatically by the `mucfgexpand` tool during the system build (see [1] for more information).

A component must verify the validity of the sinfo page by calling the appropriate validation method. In a verified SPARK/Ada component, it must be shown that this property holds before calling an actual getter function of the sinfo API.

Listing 2.3 showcases an sinfo API access in the code of the example component. The code acquires the start and end ticks of the current scheduling minor frame. This information can be used to implement a notion of relative time, or as a base for a virtual timer mechanism.

```
      Minor_Start : constant SK.Word64 := Musinfo.Instance.TSC_Schedule_Start;
      Minor_End   : constant SK.Word64 := Musinfo.Instance.TSC_Schedule_End;

```

Listing 2.3: Acquiring minor frame tick range

See section 9.4 for the sinfo API specification.

☞ It should be noted that sinfo as well as scheduling info are read-only data structures mapped into a subject's address space. Accessing this information via the respective APIs essentially results in reading the corresponding value from memory. *No direct interaction with other parts of the system, including and in particluar the Muen kernel, occurs.*

2.3 Operating System specific Methods

To allow resource discovery at runtime, the following additional methods are currently implemented for Linux VMs:

- ACPI tables
- Linux Zero-Page (ZP)

Static ACPI tables are generated by the `mugenacpi` tool and mapped into the virtual address space of a Linux guest by `mucfgexpand`.

On the Intel x86 Architecture, Linux expects a so called zero-page (ZP) with information about initramfs location, kernel command line and console information mapped into its address space. These tasks are performed by the `mugenzp` and `mucfgexpand` tool respectively.

For MirageOS/Solo5 unikernels a so called *boot info* structure is generated by the `mugensolo5` tool. It tells the unikernel how much memory it can use, the TSC frequency, command line parameters and where its application manifest is located.

Refer to the Muen System Specification [1] for more information about the involved tools.

Similar methods might be employed by other guest operating systems when porting them to Muen.

Chapter 3

VCPU Profiles

A virtual CPU (vCPU) profile controls the execution behavior of the component's virtual CPU. It allows the specification of:

- Virtualization controls
  - Runtime/Entry/Exit controls
  - Exception host/guest handling
  - CR0 host/guest ownership
  - CR4 host/guest ownership
- Access to Model-Specific Registers (MSRs)
- Initial register and segment values

VCPU profiles are assigned in the component specification at integration time and enforced by the SK. Muen differentiates two vCPU profiles which have different settings of the above:

- vCPU for native subjects
- vCPU for VM subjects

The vCPU profile of a native subject is much more restrictive, as Muen native subjects are written in SPARK/Ada with a zero-footprint (ZFP) runtime where absence of runtime errors is proven. Exception occurrence for such a component is a critical error which should be treated accordingly.

The validator tool `mucfgvalidate` ensures that the invariants for the proper execution of the Muen SK hold. This is done during integration, by verifying the vCPU profile of each subject. The kernel itself does not modify these settings during runtime. For example, it is not allowed to disable the VMX preemption timer of a subject, as this is the mechanism used by the SK to preempt subjects according to the scheduling plan.

See the Muen System Specification [1] document for the specification of all elements of a vCPU profile. The current vCPU profiles for VM and native Muen components can be found in the Appendix, section 11.1.

☞ When assigning access to MSRs, it is imperative to carefully review the potential impact the Model-Specific Register can have. This is particularly true for global MSRs that are not handled per-subject (e.g. in contrast to `IA32_EFER`). Furthermore, only a small portion of MSRs are declared as architectural, see Intel SDMs.

Chapter 4

Configuration Parameters

Components can be parameterized using the CSPEC mechanism as described in section 2.1. This mechanism generates SPARK/Ada code from a component specification in XML format. Another form of supported parameterization based on the CSPEC mechanism are the specification of configuration options. Again, this section uses the *example* component as illustration. The component was built using the following parameters.

Name	Type	Value
ahci_drv_enabled	boolean	false
This knob controls whether or not the muenblock client code is enabled in the example component.		
greeter	string	Subject running
String configuration option example		
print_serial	boolean	false
Boolean configuration option example		
print_vcpu_speed	boolean	true
Another boolean configuration option example		
serial	integer	123456789
Integer configuration option example		

As listed in the table above, the following configuration types are supported:

- Integer
- Boolean
- String

A component can use the generated constants to parameterize certain parts of the code. Listing 4.1 shows how the configuration options from the table above are used in the code of the *example* component.

```
   pragma Debug (Example_Component.Config.Print_Serial,
                 Log.Put_Line (Item => "Serial " & SK.Strings.Img
                               (SK.Word64 (Example_Component.Config.Serial))));
   pragma Debug (Example_Component.Config.Print_Vcpu_Speed,
                 Log.Put_Line (Item => "VCPU running with " & SK.Strings.Img
                               (Musinfo.Instance.TSC_Khz) & " Khz"));
```

Listing 4.1: Code parameterization using config options

The example serial number and the vCPU speed are only printed if the respective `boolean` configuration options are set to `true` in the component specification XML.[1]

[1]`pragma Debug` only executes the statement after the colon if the boolean expression before the colon is true

Chapter 5

Subject Monitoring

The subject monitoring concept consists of having a designated subject handle traps of another subject. This is useful for example to implement a trap-and-emulate approach for VM subjects or a debugger.

On execution of a trapping instruction in the monitored subject, the kernel hands over execution to the configured subject monitor (SM) to deal with the event. A potential trap is the execution of `CPUID` instruction for example[1].

Handling of such an event can range from ignoring it to fully-fledged device emulation using a device model. Execution may be handed back to the faulting subject after handling of the event.

The system policy allows the configuration of a monitor subject as seen in listing 5.1.

```
<monitor>
 <state subject="storage_linux" logical="monitor_state" virtualAddress="16#001e_0000#"
   writable="false"/>
 <loader logical="reload" subject="example" virtualAddress="16#0000#"/>
</monitor>
```

Listing 5.1: Subject Monitor (SM) configuration in the policy

The policy writer defines the access permissions to the monitored subject state. The following state items are available:

- State
  CPU register state of monitored subject
- Timed Events
  Timed events of monitored subject
- Interrupts
  Interrupts of monitored subject
- Loader
  Loader mechanism used to reset monitored subject to initial state

It is possible to define multiple state items per type, referencing different subjects. *Writable* monitoring is only allowed by subjects belonging to the same scheduling group or siblings. Read-only monitoring has no such restrictions. Furthermore, a subject may monitor its own state, e.g. interrupts, which may be useful in combination with sleeping, see 7.4.

The self-referencing `loader` element shown in listing 5.1 is part of the subject lifecycle mechanism explained in chapter 6.

The following subsections briefly introduce each state configuration, before the actual interface API is presented. See the Muen System Specification [1] for more details on the `monitor` element and its implications.

[1]CPUID always traps in VMX non-root mode

5.1 Register State

The CPU register state of the monitored subject is mapped into the monitor subject address space at the specified `virtualAddress`. The `writable` attribute defines whether write access of the monitor subject is allowed. A monitor with write access can alter the CPU register state of the monitored subject at runtime.

The SM register interface API is specified in section 9.10.

5.2 Timed Events

The timed events page of the monitored subject is mapped into the monitor subject address space at the specified `virtualAddress`. The `writable` attribute defines whether write access of the monitor subject is allowed. The programming interface is the same as for the SM subject itself, see section 9.9.

The monitored timed event mechanism is useful if the monitored subject should be preempted at certain points in time, i.e. the execution should be handed over from the monitored subject to the SM subject.

☞ Currently, the monitored subject has unconditional access to its own timed events page and could therefore clear the event set by the monitor. The mechanism is therefore not suitable to *enforce* preemption.

5.3 Interrupts

The subject interrupts page of the monitored subject is mapped into the monitor subject address space at the specified `virtualAddress`. The `writable` attribute designates whether write access by the monitor subject is allowed.

The interrupts page contains an array of four 64-bit words, which designate the 256 interrupts to inject into the monitored subject. If write access is granted, the subject monitor can use this mechanism to inject arbitrary interrupts into the monitored subject.

5.4 Loader

The memory regions of the monitored subject are mapped into the monitor subject address space at the specified `virtualAddress`. The `writable` attribute defines whether write access to these regions is allowed for the SM.

Besides potential other use-cases, the loader concept can be applied to reset a subject to its initial state, perform decoding of a trapping instruction or to (periodically) check the memory content of the monitored subject against hash sums.

See the Muen System Specification [1] for the definition of the loader element with an in-depth description of the mechanism (element `loader`, type `loaderSubjectRefType`).

Chapter 6 gives an in-depth explanation of how this concept is leveraged to implement subject reset.

5.5 By Example

This section illustrates the monitoring concept using the SM component for Linux, which is part of the Muen ecosystem of available components written in SPARK 2014. It is used to enable the execution of Linux as a VM subject on Muen.

The SM component has the CPU register state of the associated monitored Linux VM mapped. Since the SM for Muen Linux is quite simple, it only maps the CPU register state, the timed event/interrupt pages and loader functionality is not required to fulfill its task.

SM declares a package-level `State` variable of type `SK.Subject_State_Type`, as specified in section 9.10. The address of the variable in memory must be set using the Ada `Address` aspect, with the value as configured in the SM component `monitor/state` specification. The `Volatile` aspect must be used as well, to tell the compiler that the value of this variable may change outside of the programming language boundary.

Using the `State` variable, SM is able to inspect and alter the subject CPU register state of the monitored subject in response to a trap. For illustrative purposes, it is assumed that the monitored subject executes the `CPUID` instruction, which always traps in VMX non-root mode. The policy is setup so that a handover from the monitored subject to SM occurs. The following snippet shows the XML `events` section of the monitored subject:

```
<events>
 <source>
  <group name="vmx_exit">
   ...
   <event id="10" logical="cpuid" physical="trap_to_sm_1"/>
   ...
  </group>
 </source>
</events>
```

Listing 5.2: CPUID event

On `CPUID`[2] in the monitored subject, the kernel inspects the events table of the subject after exit and, according to its static configuration, hands over execution to SM.

SM first examines the exit reason of the associated subject, by using the aforementioned `State` variable:

```
      Exit_Reason := State.Exit_Reason;

      if Exit_Reason = SK.Constants.EXIT_REASON_CPUID then
         Exit_Handlers.CPUID.Process (Action => Action);
      elsif Exit_Reason = SK.Constants.EXIT_REASON_INVLPG
```

Listing 5.3: Determining the exit reason

If it concludes that the exit was caused by the execution of `CPUID`, it calls the handler for this exit reason. It does the same for other exit reasons of interest. If there is no handler for a specific exit reason, execution of the monitored subject is halted by stopping the vCPU. This is done by calling `SK.CPU.Stop`, which executes `cli; hlt` in a loop. By not returning to the monitored subject, the cooperative scheduling group is now in a halted state.

The `CPUID.Process` procedure called on a `CPUID` exit emulates a vCPU with certain features. Therefore, depending on the requested CPUID leaf in the `RAX` register, the CPU register state is updated to reflect the result of the `CPUID` instruction back to the monitored subject:

```
      RAX : constant SK.Word64 := State.Regs.RAX;
      RCX : constant SK.Word64 := State.Regs.RCX;

      Values : CPU_Values.CPUID_Values_Type;
      Res    : Boolean;
   begin
      Action          := Types.Subject_Continue;
      State.Regs.RAX := 0;
      State.Regs.RBX := 0;
      State.Regs.RCX := 0;
      State.Regs.RDX := 0;

      CPU_Values.Get_CPUID_Values
        (Leaf    => SK.Word32'Mod (RAX),
         Subleaf => SK.Byte'Mod (RCX),
         Result  => Values,
         Success => Res);
      if not Res then
         pragma Debug (Sm_Component.Config.Debug_Cpuid,
                       Debug_Ops.Put_Line
                          (Item => "Ignoring unknown CPUID leaf "
                           & SK.Strings.Img (RAX)
                           & ", subleaf " & SK.Strings.Img (RCX)));
         return;
      end if;
```

[2] Event IDs correspond to the VMX Basic Exit Reason, Intel SDM Vol. 3D, Appendix C

```
      case RAX is
         when 0 =>

            --  Cap highest valid CPUID number.

            State.Regs.RAX := 16#d#;

            State.Regs.RBX := SK.Word64 (Values.EBX);
            State.Regs.RCX := SK.Word64 (Values.ECX);
            State.Regs.RDX := SK.Word64 (Values.EDX);
         when 1 =>
```

Listing 5.4: CPUID emulation of leaf 0

After that, the SM main procedure is instructed to continue monitored subject execution by setting the `Action` out parameter of the `Process` procedure to `Subject_Continue`. The SM main procedure then resumes the monitored subject by increasing the `RIP` and calling the hypercall procedure with event ID designated by the `Events.Resume_Subject_ID` constant. This triggers a handover back to the monitored subject:

```
      case Action
      is
         when Types.Subject_Start    =>
            SK.Hypercall.Trigger_Event
              (Number => Sm_Component.Events.Resume_Subject_ID);
         when Types.Subject_Continue =>
            RIP             := State.RIP;
            Instruction_Len := State.Instruction_Len;
            State.RIP       := RIP + SK.Word64 (Instruction_Len);
            SK.Hypercall.Trigger_Event
              (Number => Sm_Component.Events.Resume_Subject_ID);
         when Types.Subject_Halt     =>
            Debug_Ops.Dump_State;
            SK.CPU.Stop;
      end case;
```

Listing 5.5: End of SM main loop

The monitored subject then continues execution at the instruction following `cpuid` in the scheduling slot of SM until the next trap occurs.

Chapter 6

Subject Lifecycle

The subject lifecycle management concept provides a lightweight mechanism to initialize and reset native subjects. It allows to set up the memory resource of a subject to a well-known state prior to the execution of subject code.

The initialization of the subject runtime environment is performed by a dedicated setup stub that runs before the effective subject code starts execution. To enable external management and synchronization, the current lifecycle state is reported via a so called *Status Page* (detailed description see 9.6). Commands are read from a *Command Page* (see 9.1). The commands are written by an external control/management subject (controller) that orchestrates all subjects under its purview.

A watchdog (WD) timer mechanism instructs a subject to update a timestamp value on the status page in regular intervals. This enables the controller subject to determine if a subject is alive or has become stuck. The watchdog value must be updated within the desired period during regular subject execution, outside of the initialization code/stub.

The current epoch value, communicated as part of the command interface, differs between subject restarts.

6.1 Policy / Config / Compilation

On the policy level, the initialization stub is treated as a library. To activate the init/reset functionality, a component simply declares a dependency in the component XML specification, see 6.3.2.

6.1.1 Memory

Read-only copies of writable memory regions and matching hashes must be specified in the subject specification. The monitor/loader mechanism with a reference to the subject itself can be used so the mappings are swapped/complemented with read-only source regions. This way all necessary regions are made reloadable by the expander.

6.1.2 Entry Point

The RIP is set in the XML specification of the `Muinit` library. Since `Mucbinsplit` will not overwrite an already present vCPU `rip` value, no other change is necessary. After initialization, a jump to the start address of the text memory region is performed to start execution of the actual subject code.

The text memory region is defined as follows:

- Logical name: text
- Type: subject_binary
- Executable: True
- Writable: False
- Content: file-backed

6.1.3 Compilation

While on the policy level, the initialization code is treated as a library, on the source level it is compiled and linked as an independent binary. This avoids mixing of text, data etc of the init and the actual component code, which is beneficial for certification. Additionally, the initialization can perform a complete reset of the component memory without interfering with its own state.

The initialization code is linked to a different address (`16#0010_0000#`) than regular Ada/SPARK component binaries (`16#0020_0000#`) to enable co-existence in the same address space.

6.2 Command and Status Interface

The initialization code implements a simple state machine by waiting for a specific, expected command, executing the associated action and then reporting completion of said action by updating the status.

A special command value `Mucontrol.Commands.CMD_SELF_CTRL` designates that the initialization code should not wait for further instructions but instead perform initialization in one go.

A watchdog interval of `Mucontrol.Commands.WD_DISABLED` designates that watchdog functionality is disabled.

For the detailed specification of the *Command Interface* see section 9.1. The *Status Interface* is specified in section 9.6.

6.3 Usage

This section describes the XML adjustments which are required to make a component / subject reloadable.

6.3.1 System Policy

Add physical memory regions to back new control and status pages.

```
...
<memory>
 <memory name="control_foobar" size="16#1000#" caching="WB">
  <fill pattern="16#00#"/>
  <hash value="none"/>
 </memory>
 <memory name="status_foobar" size="16#1000#" caching="WB">
  <hash value="none"/>
 </memory>
</memory>
...
```

Listing 6.1: Declaration of Control and Status memory regions

If a subject is placed under self-control, the fill pattern of the control page can be set to `16#ff#`. In that case the corresponding status page must have fill pattern `16#00#` to ensure proper initialization as the self-controlled subject may not access the status page at all. Since both memory regions are used during initialization, the hash of these two regions is set to none. With this, they are skipped during hash validation without having to implement some form of special-casing in the init stub.

Add a physical reset event.

```
<events>
 <event name="reset_foobar" mode="asap"/>
 ...
</events>
```

Listing 6.2: Declaration of the Reset event

6.3.2 Component Specification

Add a dependency to Muinit.

```
<component name="foo" profile="native">
 <depends>
  <library ref="muinit"/>
 </depends>
 ...
</component>
```

Listing 6.3: Dependency declaration

If a component requires access to the status and command pages during execution, i.e. beyond the initialization, also add a dependency on libmucontrol. Examples for this use case are updating the Watchdog Timer Value or Time subject of the demo system setting the runtime status to STATE_FINISHED after successfully publishing time information.

6.3.3 Subject Specification

On the subject level, the following steps are necessary:

- Add mappings for Control and Status memory regions.
- Map reset event with ID 63 to give it the highest priority. This makes sure that if a reset event is pending it will be performed upon the next entry into the subject.
- Add loader element with reference to the subject itself.

```
<subject name="foobar">
 ...
 <events>
  <target>
   <event id="63" logical="reset" physical="reset_foobar">
    <reset/>
   </event>
   ...
  </target>
 </events>
 <monitor>
  <loader subject="foobar" logical="reload" virtualAddress="16#0000#"/>
 </monitor>
 <component ref="foo">
  <map logical="control" physical="control_foobar"/>
  <map logical="status" physical="status_foobar"/>
 </component>
</subject>
```

Listing 6.4: Adjustments to Subject Specification

6.4 Operation

This section gives a description of the high-level operation.

The Subject Initialization Stub implements the finite state machine depicted in figure 6.4. Transitions are performed when the init code is in the given state and the specified command is read from the command interface. Table 6.1 lists the commands issued by the controller and the action performed by the Subject Initialization code.

☞ Since the Status Interface is writable by the potentially untrustworthy component code, its content can only be relied upon after the delivery of the reset event up until setting command to CMD_RUN and observing the status STATE_RUNNING. The reset event assures that the init code is executed and that the component cannot simply fake transitions by operating on the status page.

Command	**Action**
Sync	Set status to SYNCED and wait for next command
Erase	Zeroize writable memory regions
Prepare	Set initial content of writable memory regions by copying data from corresponding read-only regions with matching hash
Validate	Calculate and compare hashes for all memory regions that specify a hash. Report failure if a hash mismatch is detected
Run	Set initial register values, clean up stack, set status running and jump to subject code
Self-Control	Perform all initialization steps and start running subject code without further command processing

Table 6.1: Commands and their associated actions

6.4.1 Initialization

Subject initialization consists of the following steps:

1. Initialize the status interface by clearing all data and setting the state to `STATE_INITIAL`.
2. Check that Subject Info is valid. If it is not valid, set error on the status interface and return with *Success* flag set to *False*.
3. Wait for synchronization command.
4. On Failure, signal error by setting status to `DIAG_UNEXPECTED_CMD`.
5. On Success, set status to `STATE_SYNCED` and wait for either the *Erase* or *Prepare* command.
6. On Failure, signal error by setting status to `DIAG_UNEXPECTED_CMD`.
7. For every writable memory region except for the stack, the status interface, mapped subject states and timed event regions, erase their content by filling the entire region with zeros. The Sinfo index of the region currently being processed is set as diagnostics value.
8. On Success and erase command, set status to `STATE_ERASING`.
9. Erase all writable memory regions by clearing them with zeros. Then, set the status to `STATE_ERASED` and wait for the *Prepare* command.
10. On Failure, signal error by setting status to `DIAG_UNEXPECTED_CMD` and return.
11. On Success, set status to `STATE_PREPARING`.
12. Set up writable memory regions that have initial content by either copying the content from a read-only source region or filling them with a pattern.
13. On Failure, signal error and return. The diagnostics field contains the Sinfo index of the memory region that was being processed.
14. On Success, set status to `STATE_PREPARED` and wait for the Validate command.
15. On Failure, signal error by setting status to `DIAG_UNEXPECTED_CMD` and return.
16. On Success, set status to `STATE_VALIDATING`.
17. Verify hashes of *all*[1] memory regions. The entire content of each memory region is hashed and the resulting value is compared to the reference hash contained in the corresponding Sinfo entry.
18. On Failure, signal error and return. The diagnostics field contains the Sinfo index of the memory region that was being processed.

[1]This also includes read-only memory regions.

19. Otherwise, set status to `STATE_VALIDATED` and wait for the *Run* command.
20. On Failure, signal error by setting status to `DIAG_UNEXPECTED_CMD`.
21. On Success, set status to `STATE_INITIALIZING`. The final transition to `STATE_RUNNING` state is done just prior to jumping to the code of the component that has just been initialized/reset.

☞ On `CMD_SELF_CTRL` the subject may perform all of the above steps, without further synchronization/waiting for additional commands.

☞ Since the Assembly uses a a few registers to perform stack clearing etc, the registers R10-R15 will not retain their initial value. A component or subject must not make use of the initial values of these registers.

6.4.2 VM Components

For VM subjects, the same procedure as for native subjects is not viable (e.g. 2 level page tables would require IA32-e mode transition and creation of initial page tables etc.). Thus, we build on the existing loader concept. One simplification is that a loader only needs to support a single VM subject and no native subjects.

Since the subject loader (SL) in effect performs the same functionality as the initialization/reset stub for native subjects, it directly operates on the status and command pages which are associated with the VM subject. It has no status/command pages for itself. The necessary Status and Command pages are added to the loader component XML specification by declaring a library dependency on `libmucontrol`.

Since the Loader, the corresponding Linux subject as well as the associated SM are part of the same scheduling group, a form of light cooperation between SM and SL is necessary. The initialization process is as follows:

1. SM starts execution of its own initialization code. Synchronization with the controller component happens as described above. The Controller can force handover to SM by resetting Linux state.
2. After successful setup, SM enters main loop with "Invalid guest state" as Linux exit reason.
3. SM checks if Linux reset event is available. If SM Sinfo does not contain `Reset_Linux` event →go to step 9.
4. Trigger `Reset_Linux` event. This results in handover to Linux with reset target action. The initial Linux state has `CR4.VMXE = 0`, which triggers an immediate handover back to SM due to "Invalid guest state".
5. Trigger `Load_Linux` event to handover execution to SL.
6. SL syncs with controller component.
7. SL performs Linux memory initialization using the same code as the native subject initialization, reused via the Libmuinit library.
8. After successful setup of Linux memory, SL triggers a handover to SM.
9. SM inspects Linux Status Interface filled in by SL to check that no error occurred.
10. If Linux State has Error bit set, print error message and halt subject execution.
11. Check if SM is monitoring an AP Linux (`CR0.PE = 0`): wait for wakeup event.
12. SM makes Linux runnable by setting `CR4.VMXE` in Linux subject state.
13. SM triggers handover to Linux thereby starting its execution.

Except for the handover events passing on the thread of execution (without interrupt injection), SL and SM do not communicate directly.

☞ With this division of labor between SM and SL each subject has somewhat complementary privileges regarding the Linux subject: SM has access to the subject state while SL only has access to Linux's memory.

☞ SL itself does not need reset functionality since it does not keep any state. This also enables 1:1 reuse of Libmuinit just as Muinit does.

☞ SM maps the Linux Status page read-only to check for error conditions on loading Linux by SL. This avoids having an additional signalisation mechanism between SM and SL.

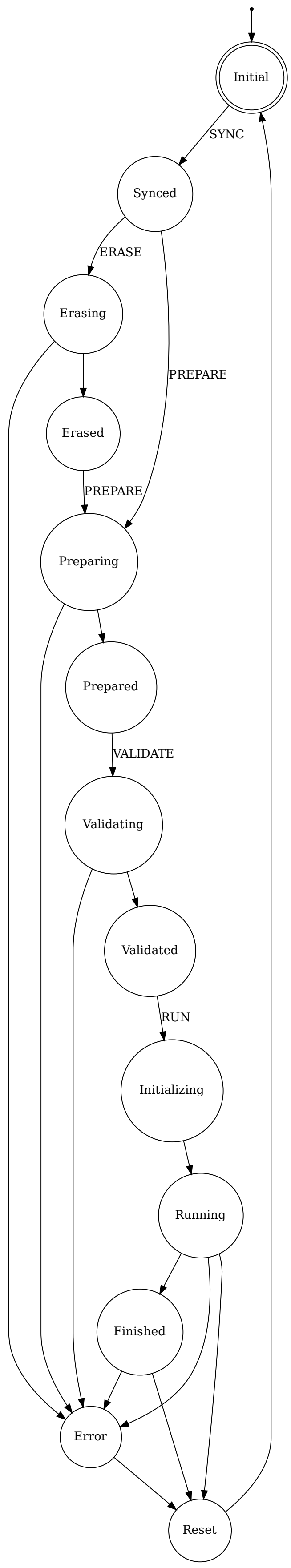

Figure 6.1: Subject Initialization State Machine

Chapter 7

Subject Yield/Sleep

Subjects which are part of the same scheduling group can do efficient, cooperative scheduling using handover events. To make more efficient use of CPU time of *scheduling partitions*, a system integrator can assign multiple scheduling groups, whose subjects do not require strict temporal isolation, to the same partition. All the scheduling groups within a partition are scheduled round robin with preemption and the opportunity to yield and/or sleep. An example for assignment to the same scheduling partition could be subjects that are event-driven or perform polling of hardware but do not process data in a steady/constant way.

7.1 Yield

Subject can yield execution for the rest of the minor frame if it does not require further CPU time. It can perform a yield operation by explicitly triggering a source event with action `subject_yield` or by causing a trap with the same action, e.g. via the `PAUSE` instruction when `PAUSE Exiting` is enabled in the subjects' vCPU profile.

When a subject yields, the kernel selects the next active scheduling group of the partition and resumes execution of the active subject of the selected group. If no other group is active, the subject which yielded will be scheduled again.

Scheduling groups of a partition are scheduled round robin. A subject that has yielded will eventually be scheduled again as soon as all other active groups of the partition have had their turn. The currently active group is changed in every new minor frame.

7.2 Sleep

Subjects which are event-driven and have no more work to do, can request to be put to sleep until they should be woken up. Such a subject can perform the sleep operation by explicitly triggering a source event with action `subject_sleep` or by causing a trap with the same action, e.g. via the `HLT` instruction when `HLT Exiting` is enabled in the subject's vCPU profile.

When a subject requests sleep, the kernel marks the scheduling group as inactive, selects the next active scheduling group of the partition and resumes execution of the active subject of the selected group. If no other group is active, the whole scheduling partition is marked as sleeping and the subject which requested to sleep is run with VMX Activity State set to `HLT`, i.e it will be scheduled but not execute any instruction.

The following events will lead to the wakeup of a a sleeping subject:

- Pending Interrupt
- Pending Target Event
- Timed Event expiry

As soon as a subject, and as a consequence its associated scheduling group, becomes active, it will be considered again whenever a new scheduling group is being selected for execution in a scheduling partition.

☞ The instruction sequence `cli; hlt;` may continue execution without issuing an interrupt when a pending interrupt is inserted. This is a deviation from Intel ISA but similar to the situation when an NMI occurs. To guard against this behavior, the instruction sequence can be put inside a (infinite) loop.

7.3 Configuration

The following XML excerpt illustrates how to configure a subject to be able to use the `PAUSE` and `HLT` instructions to trigger sleep and yield actions.

```
<events>
 ...
 <event mode="kernel" name="subject_yield"/>
 <event mode="kernel" name="subject_sleep"/>
 ..
</events>
...
<subjects>
 <subject ...>
  <vcpu>
   <vmx>
    <controls>
     <proc>
      <!-- VM-Exit on HLT instruction -->
      <HLTExiting>1</HLTExiting>
      <!-- VM-Exit on PAUSE instruction -->
      <PAUSEExiting>1</PAUSEExiting>
     </proc>
    </controls>
   </vmx>
  </vcpu>
  <events>
   <source>
    <group name="vmx_exit">
     <!-- Exit Reason 12: HLT -->
     <event id="12" logical="sleep" physical="subject_sleep">
      <subject_sleep/>
     </event>
     ...
     <!-- Exit Reason 40: PAUSE -->
     <event id="40" logical="yield" physical="subject_yield">
      <subject_yield/>
     </event>
    </group>
   </source>
  </events>
  ...
 </subject>
 ...
</subjects>
```

Listing 7.1: Declaration of Sleep and Yield events

☞ In order for these instructions to cause a VM-Exit the corresponding HLT/PAUSE Exiting vCPU controls must be set to 1. This is not necessary if yield an sleep are specified as vmcall source event actions and explicitly triggered as hypercalls.

7.4 Use case: Event-driven component

An event-driven server subject can use the sleep mechanism to avoid busy-looping and only become active when there is work pending. Producers, of which there can be multiple, inform the subject about work that it should process by sending an event with an associated interrupt. To avoid having to process each interrupt one-by-one, the server subject can map its own pending interrupts into it's address space.

First off, the subject must define a sleep event in the policy, as illustrated in in the previous section 7.3.

Then, to monitor interrupts, the following is added to the subject's `monitor` section in the system policy:

```
<monitor>
 <interrupts subject="$nameOfSubject" virtualAddress="16#....#" writable="true"/>
</monitor>
```

Listing 7.2: Declaration for monitoring ones own interrupts

With this the pending interrupts data structure used by the kernel is mapped into the subject's address space at the specified memory address.

The following code listing illustrates how the main processing loop of the server subject could be implemented.

```
Main_Loop :
loop
   if not Pending_Interrupt_Present then

      --  Go to sleep when no pending interrupts are present.

      CPU.Hlt;
   end if;

   --  Clear pending interrupts prior to processing by writing to our own
   --  interrupts page that is mapped into our address space using the
   --  monitor mechanism. This way interrupts that arrive while we are
   --  processing the ones that woke us up from sleep, will be marked as
   --  pending and not get lost.

   Clear_Pending_Interrupts;

   Process_Loop :
   loop
      Channel.Read (Data         => Buffer,
                    Data_Present => Received);
      exit when not Received;
      Process (Data => Buffer);
   end loop Process_Loop;
end loop Main_Loop;
```

Listing 7.3: Event-driven Main Loop of a native Ada/SPARK component

The subject clears all pending interrupts by writing to the monitored interrupt memory page (`Clear_Pending_Interrupts`). In conjunction with disabling interrupts via `cli` during startup, no actual interrupt injection into the subject is necessary. The only purpose that pending interrupts serve, is that a sleeping subject is woken up by the kernel when an interrupt is newly marked pending. This makes interrupt handling for subjects much more efficient in this case, since it removes the need for any subject/kernel transition per interrupt. Analogously, to determine if interrupts are pending, the subject can inspect the mapped pending interrupts page, which is represented by the `Pending_Interrupt_Present` function.

Chapter 8

Interrupt Handling

Subjects wishing to process interrupts must setup the environment according to the x86 ISA, i.e. install a Global Descriptor Table and Interrupt Descriptor Table etc. see Intel SDM Vol. 3A, section 6.12 Exception and Interrupt Handling. Native Ada/SPARK subjects can use the `SK.Interrupt_Tables` package as is illustrated by the following snippet from the Example component:

```
   SK.Interrupt_Tables.Initialize
     (Stack_Addr => Example_Component.Memory.Interrupt_Stack_Address +
        Example_Component.Memory.Interrupt_Stack_Size);
```

Listing 8.1: Initialization of Interrupt Tables

Then the component must provide a handler procedure which should be executed whenever an interrupt occurs. It must have the link name `dispatch_interrupt`. For native Ada/SPARK code, this can be achieved as follows:

```
   --  Exception/Interrupt handler.
   procedure Dispatch_Exception (Context : SK.Exceptions.Isr_Context_Type)
   with
      Export,
      Convention => C,
      Link_Name  => "dispatch_interrupt",
      Pre        => Musinfo.Instance.Is_Valid;

```

Listing 8.2: Specification of an Interrupt Handler procedure

While the interrupt handler facility takes care of saving and restoring all general purpose registers, the FPU state is not automatically handled. If the component interrupt handling code makes use of the FPU, e.g. through optimized code which leverages the wider FPU XMM registers for faster string copying, the state must be explicitly saved at the beginning of the interrupt handler and restored at the end using XSAVE/XRSTOR. Otherwise the FPU state of interrupted code is clobbered. Refer to the `Interrupt_Handler` package which illustrates how to implement interrupt handlers that may use the FPU.

Alternatively, one can choose to disallow FPU use in the interrupt handler by using the attribute `no_caller_saved_registers` in combination with the `-mgeneral-regs-only` compiler flag [1]. To apply the attribute to Ada/SPARK subprograms the following snippet can be used:

```
pragma Machine_Attribute
  (Entity         => $Subprogram_Name,
   Attribute_Name => "no_caller_saved_registers");
```

Listing 8.3: Usage of Machine attribute pragma

[1]https://gcc.gnu.org/onlinedocs/gcc/x86-Function-Attributes.html

Chapter 9

Interface API

This section specifies the interfaces of a Muen component. Where feasible, the *example* component interfaces are used to illustrate a specific interface.

A Muen component has potential access to the following interfaces:

- Shared Memory Channels
- Assigned Devices
- Subject Information (sinfo, 9.4)
- Subject Scheduling Information (schedinfo, 9.5)
- Timed Events (9.9)
- Traps (VM Exits)
- Hypercalls
- Subject Monitoring (9.10)
- Subject Lifecycle (9.1, 9.6)

Of course, it depends on the system policy of a concrete system whether a component has access to a physical device, shared memory channel or similar resources. The sinfo, schedinfo and timed event interfaces on the other hand are currently unconditionally mapped into the address space of a component.

The following sections outline each interface in detail. The initial table shows the virtual address the interface is accessible in the component, potential size and access permissions. After that, the purpose of the interface is explained before specifying the exact structure.

9.1 Mucontrol.Command.Instance.Command_Page

Type	Address
record	16#f_ffff_3000#

9.1.1 Purpose

The command page is written by a control subject and read by any subject that includes the initialization and reset functionality. Aside from the current command, the current epoch value and the watchdog timer interval are reported. Subjects should read these fields only after synchronization with the controller subject or if Command is read as `CMD_SELF_CTRL`.

9.1.2 Structure

Table 9.2: The structure of the record Mucontrol.Command.Command_Interface_Type.

Name	Type	Bytepos	First Bit	Last Bit
Command	Mucontrol.Command.Command_Type	0	0	63
Current command written by the controller subject.				
Epoch	Interfaces.Unsigned_64	8	0	63
Current epoch written by the controller subject. It is incremented on each reset of the subject.				
Watchdog_Interval	Interfaces.Unsigned_64	16	0	63
Current watchdog interval written by the controller subject. To indicate liveness to the controller, a subject must update the watchdog field within the given interval. This functionality is disabled if the field is set to WD_DISABLED.				
Reserved	Mucontrol.Command.Padding_Type	24	0	32575
Padding of the command interface to the full memory page so every bit of memory is captured by this type.				

9.2 Foo.Sender.Response

Type	Address
record	16#1_0000_2000#

9.2.1 Purpose

Example response channel. Used to illustrate a service component.

9.2.2 Structure

Table 9.4: The structure of the record Foo.Message_Type.

Name	Type	Bytepos	First Bit	Last Bit
Size	SK.Word16	0	0	15
Example modular type field, designating the size of data.				
Data	Foo.Data_Array	2	0	16383
Example array of bytes.				

9.3 Foo.Receiver.Request

Type	Address
record	16#1_0000_1000#

9.3.1 Purpose

Example request channel. Used to illustrate a service component.

9.3.2 Structure

Table 9.6: The structure of the record Foo.Message_Type.

Name	Type	Bytepos	First Bit	Last Bit
Size	SK.Word16	0	0	15
Example modular type field, designating the size of data.				
Data	Foo.Data_Array	2	0	16383
Example array of bytes.				

9.4 Musinfo.Instance.Object

Type	Address
record	16#e_0000_0000#

9.4.1 Purpose

A subject information record provides means to retrieve information about the execution environment.

Subject resources are exported as variant records, which are all explicitly padded in order to guarantee an exact layout and proper initialization of unused space in smaller variants. The padding size of each variant is determined by the size of the largest variant.

See the `musinfo.ads` SPARK/Ada specification file for the exact layout of each record variant.

9.4.2 Structure

Table 9.8: The structure of the record Musinfo.Subject_Info_Type.

Name	Type	Bytepos	First Bit	Last Bit
Magic	Interfaces.Unsigned_64	0	0	63
Sinfo magic, used to check validity of memory region				
TSC_Khz	Musinfo.TSC_Tick_Rate_Khz_Type	8	0	31
Tick rate of VCPU in Khz				
Name	Musinfo.Name_Type	12	0	519
Subject name				
Resource_Count	Interfaces.Unsigned_16	77	0	15
Number of active resource records				
Padding	Interfaces.Unsigned_8	79	0	7
8-bit padding				
Resources	Musinfo.Resource_Array	80	0	261119
Array of resource records.				

9.5 Musinfo.Instance.Sched_Info

Type	Address
record	16#e_0000_8000#

9.5.1 Purpose

The Subject Scheduling Information (schedinfo) mechanism exports coarse grained scheduling information to subjects. More specifically, the start and end ticks of the current minor frame are exported.

9.5.2 Structure

Table 9.10: The structure of the record Muschedinfo.Scheduling_Info_Type.

Name	Type	Bytepos	First Bit	Last Bit
TSC_Schedule_Start	Interfaces.Unsigned_64	0	0	63
Tick value of minor frame start				
TSC_Schedule_End	Interfaces.Unsigned_64	8	0	63
Tick value of minor frame end				

9.6 Mucontrol.Status.Instance.Status_Page

Type	Address
record	16#f_ffff_2000#

9.6.1 Purpose

The status page is written by a subject that includes the initialization and reset functionality and read by the control subject. It is used to report the current state of the initialization/reset process. Subjects may use unreserved state numbers to indicate custom runtime information. As an example: a subject may set state to `16#1000#` after some information has been written to a shared memory region to indicate availability of said information.

An error is designated by the most significant bit of State. If it is set, then an error condition is present. By oring `STATE_ERROR`, the current state value is preserved, which can be helpful for debugging purposes since it directly designates the failure state.

The watchdog field is used to report liveliness of the subject by writing a new timestamp value within the WD interval specified on the command page. The diagnostics field can be used to report additional debug information, e.g. when transitioning to an error state.

9.6.2 Structure

Table 9.12: The structure of the record Mucontrol.Status.Status_Interface_Type.

Name	Type	Bytepos	First Bit	Last Bit
State	Mucontrol.Status.State_Type	0	0	63
Current state reported by the subject.				
Watchdog	Interfaces.Unsigned_64	8	0	63
Current watchdog timestamp reported by the subject.				
Diagnostics	Mucontrol.Status.Diagnostics_Type	16	0	63
Current diagnostics value reported by the subject. Its meaning depends on the current state.				
Reserved	Mucontrol.Status.Padding_Type	24	0	32575
Padding of the status interface to the full memory page so every bit of memory is captured by this type.				

9.7 Debuglog.Sink.Message_Channel

Type	Address
Debuglog.Sink.CT	16#f_fff0_0000#

9.7.1 Purpose

Shared memory channel to the debug server (dbgserver) subject. Implemented using the writer provided by the Libmuchannel library. Components may use Libmudebuglog Debuglog.Client.Put* operations to transfer logging information to the dbgserver.

9.8 Hypercalls

The SK.Hypercall package is used to trigger hypercalls into the Muen SK. Components are only able to trigger events which are defined in the system policy. The Trigger_Event procedure triggers a hypercall given by the Number argument. Internally, the `vmcall` instruction with the event number in register RAX is used on the x86_64 architecture to initiate a hypercall into the kernel.

9.9 Timed Events

Type	Address
record	16#e_0001_0000#

9.9.1 Purpose

Timed events allow a subject to trigger a policy defined event at a given CPU tick count.

This is useful to implement synthetic timers using the event injection mechanism of the kernel, among other things.

9.9.2 Structure

Table 9.15: The structure of the record Mutimedevents.Timed__Event__Interface__Type.

Name	Type	Bytepos	First Bit	Last Bit
TSC_Trigger_Value	Interfaces.Unsigned_64	0	0	63
CPU tick count to fire the event designated by the event number field.				
Event_Nr	Mutimedevents.Unsigned_6	8	0	5
Number of event to trigger.				
Padding	Mutimedevents.Padding_Type	8	6	63
❦No documentation available.				

9.10 Monitored Subject Register State

Type	Address
record	16#1e_0000#

9.10.1 Purpose

Access to the subject register state of a monitored subject.

9.10.2 Structure

Table 9.17: The structure of the record SK.Subject__State__Type.

Name	Type	Bytepos	First Bit	Last Bit
Regs	SK.CPU_Registers_Type	0	0	1023
CPU registers CR2, RAX, RBX, RCX, RDX, RDI, RSI, RBP, R08-R15 (64 bits each).				
Exit_Reason	SK.Word32	128	0	31
Exit reason; Intel SDM Vol. 3C, "24.9.1 Basic VM-Exit Information". This field encodes the reason for the VM exit.				
Intr_State	SK.Word32	132	0	31
Interruptibility state; Intel SDM Vol. 3C, "24.4.2 Guest Non-Register State". The IA-32 architecture includes features that permit certain events to be blocked for a period of time. This field contains information about such blocking.				
Activity_State	SK.Word32	136	0	31
Guest activity state; Intel SDM Vol. 3C, "24.4.2 Guest Non-Register State". This field identifies the logical processor's activity state.				
SYSENTER_CS	SK.Word32	140	0	31
Guest IA32_SYSENTER_CS MSR; Intel SDM Vol. 3C, "24.4.1 Guest Register State".				
Instruction_Len	SK.Word32	144	0	31
Exit instruction length; Intel SDM Vol. 3C, "24.9.4 Information for VM Exits Due to Instruction Execution". This field receives the length in bytes of the instruction whose execution led to the VM exit. Also used in the context of software interrupts or software exceptions.				
Exit_Qualification	SK.Word64	148	0	63
Exit qualification; Intel SDM Vol. 3C, "24.9.1 Basic VM-Exit Information". This field contains additional information about the cause of VM exits.				
Guest_Phys_Addr	SK.Word64	156	0	63
Guest-physical address of exit due to EPT violations and EPT misconfigurations; Intel SDM Vol. 3C, "24.9.1 Basic VM-Exit Information".				
RIP	SK.Word64	164	0	63
Guest RIP register; Intel SDM Vol. 3C, "24.4.1 Guest Register State".				
RSP	SK.Word64	172	0	63
Guest RSP register; Intel SDM Vol. 3C, "24.4.1 Guest Register State".				
CR0	SK.Word64	180	0	63
Guest CR0 control register; Intel SDM Vol. 3C, "24.4.1 Guest Register State".				
SHADOW_CR0	SK.Word64	188	0	63
CR0 control register read shadow; Intel SDM Vol. 3C, "24.6.6 Guest/Host Masks and Read Shadows for CR0 and CR4".				
CR3	SK.Word64	196	0	63
Guest CR3 control register.				
CR4	SK.Word64	204	0	63
Guest CR4 control register; Intel SDM Vol. 3C, "24.4.1 Guest Register State".				

(continuation)

Name	Type	Bytepos	First Bit	Last Bit
SHADOW_CR4	SK.Word64	212	0	63
CR4 control register read shadow; Intel SDM Vol. 3C, "24.6.6 Guest/Host Masks and Read Shadows for CR0 and CR4".				
RFLAGS	SK.Word64	220	0	63
Guest RFLAGS register; Intel SDM Vol. 3C, "24.4.1 Guest Register State".				
IA32_EFER	SK.Word64	228	0	63
Guest IA32_EFER MSR; Intel SDM Vol. 3C, "24.4.1 Guest Register State".				
SYSENTER_ESP	SK.Word64	236	0	63
Guest IA32_SYSENTER_ESP MSR; Intel SDM Vol. 3C, "24.4.1 Guest Register State".				
SYSENTER_EIP	SK.Word64	244	0	63
Guest IA32_SYSENTER_EIP MSR; Intel SDM Vol. 3C, "24.4.1 Guest Register State".				
Segment_Regs	SK.Segment_Registers_Type	252	0	1535
Guest segment registers CS, SS, DS, ES, FS, GS, TR and LDTR. Intel SDM Vol. 3C, "24.4.1 Guest Register State".				
GDTR	SK.Segment_Type	444	0	191
Guest global descriptor table register (GDTR).				
IDTR	SK.Segment_Type	468	0	191
Guest interrupt descriptor table register (IDTR).				
Running	Standard.Boolean	492	0	0
Flag used by the kernel to track whether the subject is running or sleeping.				
Padding	SK.Reserved_Type	492	1	7
❦No documentation available.				

Chapter 10

Verification Conditions Summary (SPARK 2014)

Muen is very well suited to work with verified SPARK/Ada components. The native vCPU profile is used to run such components as guest on top of the kernel. The table below shows the verification results for the example component.

	Total	Flow	CodePeer	Provers	Justified	Unproved
Data Dependencies	43	43	0	0	0	0
Flow Dependencies	33	33	0	0	0	0
Initialization	53	53	0	0	0	0
Non Aliasing	1	1	0	0	0	0
Runtime Checks	141	0	0	140	1	0
Assertions	4	0	0	4	0	0
Functional Contracts	77	0	0	77	0	0
LSP Verification	0	0	0	0	0	0
Totals	354	132	0	221	1	0

Chapter 11

Appendix

11.1 VCPU Profiles

```
<?xml version="1.0"?>
<vcpu>
 <vmx>
  <controls>
   <pin>
    <ExternalInterruptExiting>1</ExternalInterruptExiting>
    <NMIExiting>1</NMIExiting>
    <VirtualNMIs>0</VirtualNMIs>
    <ActivateVMXTimer>1</ActivateVMXTimer>
    <ProcessPostedInterrupts>0</ProcessPostedInterrupts>
   </pin>
   <proc>
    <InterruptWindowExiting>0</InterruptWindowExiting>
    <UseTSCOffsetting>0</UseTSCOffsetting>
    <HLTExiting>0</HLTExiting>
    <INVLPGExiting>1</INVLPGExiting>
    <MWAITExiting>1</MWAITExiting>
    <RDPMCExiting>1</RDPMCExiting>
    <RDTSCExiting>1</RDTSCExiting>
    <CR3LoadExiting>1</CR3LoadExiting>
    <CR3StoreExiting>1</CR3StoreExiting>
    <CR8LoadExiting>1</CR8LoadExiting>
    <CR8StoreExiting>1</CR8StoreExiting>
    <UseTPRShadow>0</UseTPRShadow>
    <NMIWindowExiting>0</NMIWindowExiting>
    <MOVDRExiting>1</MOVDRExiting>
    <UnconditionalIOExiting>0</UnconditionalIOExiting>
    <UseIOBitmaps>1</UseIOBitmaps>
    <MonitorTrapFlag>0</MonitorTrapFlag>
    <UseMSRBitmaps>1</UseMSRBitmaps>
    <MONITORExiting>1</MONITORExiting>
    <PAUSEExiting>0</PAUSEExiting>
    <Activate2ndaryControls>1</Activate2ndaryControls>
   </proc>
   <proc2>
    <VirtualAPICAccesses>0</VirtualAPICAccesses>
    <EnableEPT>0</EnableEPT>
    <DescriptorTableExiting>0</DescriptorTableExiting>
    <EnableRDTSCP>0</EnableRDTSCP>
    <Virtualizex2APICMode>0</Virtualizex2APICMode>
    <EnableVPID>0</EnableVPID>
    <WBINVDExiting>1</WBINVDExiting>
    <UnrestrictedGuest>0</UnrestrictedGuest>
    <APICRegisterVirtualization>0</APICRegisterVirtualization>
    <VirtualInterruptDelivery>0</VirtualInterruptDelivery>
    <PAUSELoopExiting>0</PAUSELoopExiting>
    <RDRANDExiting>0</RDRANDExiting>
    <EnableINVPCID>0</EnableINVPCID>
    <EnableVMFunctions>0</EnableVMFunctions>
   </proc2>
   <entry>
    <LoadDebugControls>0</LoadDebugControls>
    <IA32eModeGuest>1</IA32eModeGuest>
    <EntryToSMM>0</EntryToSMM>
    <DeactiveDualMonitorTreatment>0</DeactiveDualMonitorTreatment>
    <LoadIA32PERFGLOBALCTRL>0</LoadIA32PERFGLOBALCTRL>
    <LoadIA32PAT>0</LoadIA32PAT>
    <LoadIA32EFER>0</LoadIA32EFER>
   </entry>
   <exit>
    <SaveDebugControls>0</SaveDebugControls>
    <HostAddressspaceSize>1</HostAddressspaceSize>
    <LoadIA32PERFGLOBALCTRL>0</LoadIA32PERFGLOBALCTRL>
    <AckInterruptOnExit>1</AckInterruptOnExit>
    <SaveIA32PAT>0</SaveIA32PAT>
    <LoadIA32PAT>0</LoadIA32PAT>
    <SaveIA32EFER>0</SaveIA32EFER>
    <LoadIA32EFER>0</LoadIA32EFER>
    <SaveVMXTimerValue>0</SaveVMXTimerValue>
   </exit>
  </controls>
  <masks>
   <exception>
    <DivideError>1</DivideError>
    <Debug>1</Debug>
    <Breakpoint>1</Breakpoint>
    <Overflow>1</Overflow>
    <BOUNDRangeExceeded>1</BOUNDRangeExceeded>
    <InvalidOpcode>1</InvalidOpcode>
    <DeviceNotAvailable>1</DeviceNotAvailable>
    <DoubleFault>1</DoubleFault>
    <CoprocessorSegmentOverrun>1</CoprocessorSegmentOverrun>
    <InvalidTSS>1</InvalidTSS>
    <SegmentNotPresent>1</SegmentNotPresent>
    <StackSegmentFault>1</StackSegmentFault>
    <GeneralProtection>1</GeneralProtection>
    <PageFault>1</PageFault>
    <x87FPUFloatingPointError>1</x87FPUFloatingPointError>
    <AlignmentCheck>1</AlignmentCheck>
    <MachineCheck>1</MachineCheck>
    <SIMDFloatingPointException>1</SIMDFloatingPointException>
   </exception>
   <cr0>
    <ProtectionEnable>1</ProtectionEnable>
    <MonitorCoprocessor>1</MonitorCoprocessor>
    <Emulation>1</Emulation>
    <TaskSwitched>1</TaskSwitched>
    <ExtensionType>1</ExtensionType>
    <NumericError>1</NumericError>
    <WriteProtect>1</WriteProtect>
    <AlignmentMask>1</AlignmentMask>
     <!-- WARNING: Do not unmask CR0.NW(29) and CR0.CD(30) -->
     <!--          as these bits are not handled by VMX    -->
    <NotWritethrough>1</NotWritethrough>
    <CacheDisable>1</CacheDisable>
    <Paging>1</Paging>
   </cr0>
   <cr4>
    <Virtual8086>1</Virtual8086>
    <ProtectedVirtualInts>1</ProtectedVirtualInts>
    <TimeStampDisable>1</TimeStampDisable>
    <DebuggingExtensions>1</DebuggingExtensions>
    <PageSizeExtensions>1</PageSizeExtensions>
    <PhysicalAddressExtension>1</PhysicalAddressExtension>
    <MachineCheckEnable>1</MachineCheckEnable>
    <PageGlobalEnable>1</PageGlobalEnable>
    <PerfCounterEnable>1</PerfCounterEnable>
    <OSSupportFXSAVE>1</OSSupportFXSAVE>
    <OSSupportSIMDExceptions>1</OSSupportSIMDExceptions>
    <UMInstructionPrevention>1</UMInstructionPrevention>
    <VMXEnable>1</VMXEnable>
    <SMXEnable>1</SMXEnable>
    <FSGSBASEEnable>1</FSGSBASEEnable>
    <PCIDEnable>1</PCIDEnable>
    <XSAVEEnable>1</XSAVEEnable>
    <SMEPEnable>1</SMEPEnable>
    <SMAPEnable>1</SMAPEnable>
    <ProtectionKeyEnable>1</ProtectionKeyEnable>
   </cr4>
  </masks>
 </vmx>
 <msrs/>
 <registers>
  <gpr>
   <rip>16#0020_0000#</rip>
   <rsp>16#3000#</rsp>
   <rax>16#0000#</rax>
   <rbx>16#0000#</rbx>
   <rcx>16#0000#</rcx>
   <rdx>16#0000#</rdx>
   <rdi>16#0000#</rdi>
   <rsi>16#0000#</rsi>
   <rbp>16#0000#</rbp>
   <r08>16#0000#</r08>
   <r09>16#0000#</r09>
   <r10>16#0000#</r10>
   <r11>16#0000#</r11>
   <r12>16#0000#</r12>
   <r13>16#0000#</r13>
   <r14>16#0000#</r14>
   <r15>16#0000#</r15>
  </gpr>
  <cr0>
   <ProtectionEnable>1</ProtectionEnable>
   <MonitorCoprocessor>1</MonitorCoprocessor>
   <Emulation>0</Emulation>
   <TaskSwitched>0</TaskSwitched>
   <ExtensionType>1</ExtensionType>
   <NumericError>1</NumericError>
   <WriteProtect>1</WriteProtect>
   <AlignmentMask>0</AlignmentMask>
   <NotWritethrough>0</NotWritethrough>
   <CacheDisable>0</CacheDisable>
   <Paging>1</Paging>
  </cr0>
  <cr0Shadow>
   <ProtectionEnable>1</ProtectionEnable>
   <MonitorCoprocessor>1</MonitorCoprocessor>
   <Emulation>0</Emulation>
   <TaskSwitched>0</TaskSwitched>
   <ExtensionType>1</ExtensionType>
   <NumericError>1</NumericError>
   <WriteProtect>1</WriteProtect>
   <AlignmentMask>0</AlignmentMask>
   <NotWritethrough>0</NotWritethrough>
   <CacheDisable>0</CacheDisable>
   <Paging>1</Paging>
  </cr0Shadow>
  <cr4>
   <Virtual8086>0</Virtual8086>
   <ProtectedVirtualInts>0</ProtectedVirtualInts>
   <TimeStampDisable>0</TimeStampDisable>
   <DebuggingExtensions>0</DebuggingExtensions>
   <PageSizeExtensions>0</PageSizeExtensions>
   <PhysicalAddressExtension>1</PhysicalAddressExtension>
   <MachineCheckEnable>1</MachineCheckEnable>
   <PageGlobalEnable>0</PageGlobalEnable>
   <PerfCounterEnable>0</PerfCounterEnable>
   <OSSupportFXSAVE>1</OSSupportFXSAVE>
   <OSSupportSIMDExceptions>0</OSSupportSIMDExceptions>
   <UMInstructionPrevention>0</UMInstructionPrevention>
   <VMXEnable>1</VMXEnable>
   <SMXEnable>0</SMXEnable>
   <FSGSBASEEnable>0</FSGSBASEEnable>
   <PCIDEnable>0</PCIDEnable>
   <XSAVEEnable>0</XSAVEEnable>
   <SMEPEnable>0</SMEPEnable>
```

```
    <SMAPEnable>0</SMAPEnable>
    <ProtectionKeyEnable>0</ProtectionKeyEnable>
   </cr4>
   <cr4Shadow>
    <Virtual8086>0</Virtual8086>
    <ProtectedVirtualInts>0</ProtectedVirtualInts>
    <TimeStampDisable>0</TimeStampDisable>
    <DebuggingExtensions>0</DebuggingExtensions>
    <PageSizeExtensions>0</PageSizeExtensions>
    <PhysicalAddressExtension>1</PhysicalAddressExtension>
    <MachineCheckEnable>1</MachineCheckEnable>
    <PageGlobalEnable>0</PageGlobalEnable>
    <PerfCounterEnable>0</PerfCounterEnable>
    <OSSupportFXSAVE>1</OSSupportFXSAVE>
    <OSSupportSIMDExceptions>0</OSSupportSIMDExceptions>
    <UMInstructionPrevention>0</UMInstructionPrevention>
    <VMXEnable>1</VMXEnable>
    <SMXEnable>0</SMXEnable>
    <FSGSBASEEnable>0</FSGSBASEEnable>
    <PCIDEnable>0</PCIDEnable>
    <XSAVEEnable>0</XSAVEEnable>
    <SMEPEnable>0</SMEPEnable>
    <SMAPEnable>0</SMAPEnable>
    <ProtectionKeyEnable>0</ProtectionKeyEnable>
   </cr4Shadow>
   <segments>
    <cs access="16#a09b#" base="16#0000#" limit="16#ffff_ffff#" selector="16#0008#"/>
    <ds access="16#c093#" base="16#0000#" limit="16#ffff_ffff#" selector="16#0010#"/>
    <es access="16#c093#" base="16#0000#" limit="16#ffff_ffff#" selector="16#0010#"/>
    <fs access="16#0001_0000#" base="16#0000#" limit="16#0000#" selector="16#0000#"/>
    <gs access="16#0001_0000#" base="16#0000#" limit="16#0000#" selector="16#0000#"/>
    <ss access="16#c093#" base="16#0000#" limit="16#ffff_ffff#" selector="16#0010#"/>
    <tr access="16#008b#" base="16#0000#" limit="16#ffff#" selector="16#0018#"/>
    <ldtr access="16#0001_0000#" base="16#0000#" limit="16#0000#" selector="16#0000#"/>
   </segments>
  </registers>
</vcpu>
```

Listing 11.1: Native vCPU

```
<?xml version="1.0"?>
<vcpu>
 <vmx>
  <controls>
   <pin>
    <ExternalInterruptExiting>1</ExternalInterruptExiting>
    <NMIExiting>1</NMIExiting>
    <VirtualNMIs>0</VirtualNMIs>
    <ActivateVMXTimer>1</ActivateVMXTimer>
    <ProcessPostedInterrupts>0</ProcessPostedInterrupts>
   </pin>
   <proc>
    <InterruptWindowExiting>0</InterruptWindowExiting>
    <UseTSCOffsetting>0</UseTSCOffsetting>
    <HLTExiting>0</HLTExiting>
    <INVLPGExiting>1</INVLPGExiting>
    <MWAITExiting>1</MWAITExiting>
    <RDPMCExiting>1</RDPMCExiting>
    <RDTSCExiting>1</RDTSCExiting>
    <CR3LoadExiting>0</CR3LoadExiting>
    <CR3StoreExiting>0</CR3StoreExiting>
    <CR8LoadExiting>1</CR8LoadExiting>
    <CR8StoreExiting>1</CR8StoreExiting>
    <UseTPRShadow>0</UseTPRShadow>
    <NMIWindowExiting>0</NMIWindowExiting>
    <MOVDRExiting>1</MOVDRExiting>
    <UnconditionalIOExiting>0</UnconditionalIOExiting>
    <UseIOBitmaps>1</UseIOBitmaps>
    <MonitorTrapFlag>0</MonitorTrapFlag>
    <UseMSRBitmaps>1</UseMSRBitmaps>
    <MONITORExiting>1</MONITORExiting>
    <PAUSEExiting>0</PAUSEExiting>
    <Activate2ndaryControls>1</Activate2ndaryControls>
```

```xml
  </proc>
  <proc2>
   <VirtualAPICAccesses>0</VirtualAPICAccesses>
   <EnableEPT>1</EnableEPT>
   <DescriptorTableExiting>0</DescriptorTableExiting>
   <EnableRDTSCP>0</EnableRDTSCP>
   <Virtualizex2APICMode>0</Virtualizex2APICMode>
   <EnableVPID>0</EnableVPID>
   <WBINVDExiting>1</WBINVDExiting>
   <UnrestrictedGuest>1</UnrestrictedGuest>
   <APICRegisterVirtualization>0</APICRegisterVirtualization>
   <VirtualInterruptDelivery>0</VirtualInterruptDelivery>
   <PAUSELoopExiting>0</PAUSELoopExiting>
   <RDRANDExiting>0</RDRANDExiting>
   <EnableINVPCID>0</EnableINVPCID>
   <EnableVMFunctions>0</EnableVMFunctions>
  </proc2>
  <entry>
   <LoadDebugControls>0</LoadDebugControls>
   <IA32eModeGuest>0</IA32eModeGuest>
   <EntryToSMM>0</EntryToSMM>
   <DeactiveDualMonitorTreatment>0</DeactiveDualMonitorTreatment>
   <LoadIA32PERFGLOBALCTRL>0</LoadIA32PERFGLOBALCTRL>
   <LoadIA32PAT>0</LoadIA32PAT>
   <LoadIA32EFER>1</LoadIA32EFER>
  </entry>
  <exit>
   <SaveDebugControls>0</SaveDebugControls>
   <HostAddressspaceSize>1</HostAddressspaceSize>
   <LoadIA32PERFGLOBALCTRL>0</LoadIA32PERFGLOBALCTRL>
   <AckInterruptOnExit>1</AckInterruptOnExit>
   <SaveIA32PAT>0</SaveIA32PAT>
   <LoadIA32PAT>0</LoadIA32PAT>
   <SaveIA32EFER>1</SaveIA32EFER>
   <LoadIA32EFER>1</LoadIA32EFER>
   <SaveVMXTimerValue>0</SaveVMXTimerValue>
  </exit>
 </controls>
 <masks>
  <exception>
   <DivideError>0</DivideError>
   <Debug>0</Debug>
   <Breakpoint>0</Breakpoint>
   <Overflow>0</Overflow>
   <BOUNDRangeExceeded>0</BOUNDRangeExceeded>
   <InvalidOpcode>0</InvalidOpcode>
   <DeviceNotAvailable>0</DeviceNotAvailable>
   <DoubleFault>0</DoubleFault>
   <CoprocessorSegmentOverrun>0</CoprocessorSegmentOverrun>
   <InvalidTSS>0</InvalidTSS>
   <SegmentNotPresent>0</SegmentNotPresent>
   <StackSegmentFault>0</StackSegmentFault>
   <GeneralProtection>0</GeneralProtection>
   <PageFault>0</PageFault>
   <x87FPUFloatingPointError>0</x87FPUFloatingPointError>
   <AlignmentCheck>0</AlignmentCheck>
   <MachineCheck>1</MachineCheck>
   <SIMDFloatingPointException>0</SIMDFloatingPointException>
  </exception>
  <cr0>
   <ProtectionEnable>0</ProtectionEnable>
   <MonitorCoprocessor>0</MonitorCoprocessor>
   <Emulation>0</Emulation>
   <TaskSwitched>0</TaskSwitched>
    <!-- CR0.ET(4) is ignored on all supported CPUs -->
   <ExtensionType>0</ExtensionType>
   <NumericError>1</NumericError>
   <WriteProtect>0</WriteProtect>
   <AlignmentMask>0</AlignmentMask>
    <!-- WARNING: Do not unmask CR0.NW(29) and CR0.CD(30) -->
    <!--          as these bits are not handled by VMX    -->
   <NotWritethrough>1</NotWritethrough>
   <CacheDisable>1</CacheDisable>
```

```xml
     <Paging>0</Paging>
    </cr0>
    <cr4>
     <Virtual8086>0</Virtual8086>
     <ProtectedVirtualInts>0</ProtectedVirtualInts>
     <TimeStampDisable>0</TimeStampDisable>
     <DebuggingExtensions>0</DebuggingExtensions>
     <PageSizeExtensions>0</PageSizeExtensions>
     <PhysicalAddressExtension>0</PhysicalAddressExtension>
     <MachineCheckEnable>1</MachineCheckEnable>
     <PageGlobalEnable>0</PageGlobalEnable>
     <PerfCounterEnable>0</PerfCounterEnable>
     <OSSupportFXSAVE>0</OSSupportFXSAVE>
     <OSSupportSIMDExceptions>0</OSSupportSIMDExceptions>
     <UMInstructionPrevention>0</UMInstructionPrevention>
     <VMXEnable>1</VMXEnable>
     <SMXEnable>0</SMXEnable>
     <FSGSBASEEnable>0</FSGSBASEEnable>
     <PCIDEnable>0</PCIDEnable>
     <XSAVEEnable>0</XSAVEEnable>
     <SMEPEnable>0</SMEPEnable>
     <SMAPEnable>0</SMAPEnable>
     <ProtectionKeyEnable>1</ProtectionKeyEnable>
    </cr4>
   </masks>
  </vmx>
  <msrs>
   <!-- IA32_SYSENTER_CS/ESP/EIP -->
   <msr start="16#0174#" end="16#0176#" mode="rw"/>
   <!-- IA32_EFER/STAR/LSTAR/CSTAR/FMASK -->
   <msr start="16#c000_0080#" end="16#c000_0084#" mode="rw"/>
   <!-- IA32_FS_BASE/GS_BASE/KERNEL_GS_BASE -->
   <msr start="16#c000_0100#" end="16#c000_0102#" mode="rw"/>
  </msrs>
  <registers>
   <gpr>
    <rip>16#0040_0000#</rip>
    <rsp>16#0000#</rsp>
    <rax>16#0000#</rax>
    <rbx>16#0000#</rbx>
    <rcx>16#0000#</rcx>
    <rdx>16#0000#</rdx>
    <rdi>16#0000#</rdi>
    <rsi>16#0000#</rsi>
    <rbp>16#0000#</rbp>
    <r08>16#0000#</r08>
    <r09>16#0000#</r09>
    <r10>16#0000#</r10>
    <r11>16#0000#</r11>
    <r12>16#0000#</r12>
    <r13>16#0000#</r13>
    <r14>16#0000#</r14>
    <r15>16#0000#</r15>
   </gpr>
   <cr0>
    <ProtectionEnable>1</ProtectionEnable>
    <MonitorCoprocessor>0</MonitorCoprocessor>
    <Emulation>1</Emulation>
    <TaskSwitched>0</TaskSwitched>
    <ExtensionType>1</ExtensionType>
    <NumericError>1</NumericError>
    <WriteProtect>0</WriteProtect>
    <AlignmentMask>0</AlignmentMask>
    <NotWritethrough>0</NotWritethrough>
    <CacheDisable>0</CacheDisable>
    <Paging>0</Paging>
   </cr0>
   <cr0Shadow>
    <ProtectionEnable>1</ProtectionEnable>
    <MonitorCoprocessor>0</MonitorCoprocessor>
    <Emulation>0</Emulation>
    <TaskSwitched>0</TaskSwitched>
    <ExtensionType>0</ExtensionType>
    <NumericError>1</NumericError>
    <WriteProtect>0</WriteProtect>
    <AlignmentMask>0</AlignmentMask>
    <NotWritethrough>0</NotWritethrough>
    <CacheDisable>0</CacheDisable>
    <Paging>0</Paging>
   </cr0Shadow>
   <cr4>
    <Virtual8086>0</Virtual8086>
    <ProtectedVirtualInts>0</ProtectedVirtualInts>
    <TimeStampDisable>0</TimeStampDisable>
    <DebuggingExtensions>0</DebuggingExtensions>
    <PageSizeExtensions>0</PageSizeExtensions>
    <PhysicalAddressExtension>1</PhysicalAddressExtension>
    <MachineCheckEnable>1</MachineCheckEnable>
    <PageGlobalEnable>0</PageGlobalEnable>
    <PerfCounterEnable>0</PerfCounterEnable>
    <OSSupportFXSAVE>0</OSSupportFXSAVE>
    <OSSupportSIMDExceptions>0</OSSupportSIMDExceptions>
    <UMInstructionPrevention>0</UMInstructionPrevention>
    <VMXEnable>1</VMXEnable>
    <SMXEnable>0</SMXEnable>
    <FSGSBASEEnable>0</FSGSBASEEnable>
    <PCIDEnable>0</PCIDEnable>
    <XSAVEEnable>0</XSAVEEnable>
    <SMEPEnable>0</SMEPEnable>
    <SMAPEnable>0</SMAPEnable>
    <ProtectionKeyEnable>0</ProtectionKeyEnable>
   </cr4>
   <cr4Shadow>
    <Virtual8086>0</Virtual8086>
    <ProtectedVirtualInts>0</ProtectedVirtualInts>
    <TimeStampDisable>0</TimeStampDisable>
    <DebuggingExtensions>0</DebuggingExtensions>
    <PageSizeExtensions>0</PageSizeExtensions>
    <PhysicalAddressExtension>0</PhysicalAddressExtension>
    <MachineCheckEnable>0</MachineCheckEnable>
    <PageGlobalEnable>0</PageGlobalEnable>
    <PerfCounterEnable>0</PerfCounterEnable>
    <OSSupportFXSAVE>0</OSSupportFXSAVE>
    <OSSupportSIMDExceptions>0</OSSupportSIMDExceptions>
    <UMInstructionPrevention>0</UMInstructionPrevention>
    <VMXEnable>0</VMXEnable>
    <SMXEnable>0</SMXEnable>
    <FSGSBASEEnable>0</FSGSBASEEnable>
    <PCIDEnable>0</PCIDEnable>
    <XSAVEEnable>0</XSAVEEnable>
    <SMEPEnable>0</SMEPEnable>
    <SMAPEnable>0</SMAPEnable>
    <ProtectionKeyEnable>0</ProtectionKeyEnable>
   </cr4Shadow>
   <segments>
    <cs access="16#c09b#" base="16#0000#" limit="16#ffff_ffff#" selector="16#0008#"/>
    <ds access="16#c093#" base="16#0000#" limit="16#ffff_ffff#" selector="16#0010#"/>
    <es access="16#c093#" base="16#0000#" limit="16#ffff_ffff#" selector="16#0010#"/>
    <fs access="16#0001_0000#" base="16#0000#" limit="16#0000#" selector="16#0000#"/>
    <gs access="16#0001_0000#" base="16#0000#" limit="16#0000#" selector="16#0000#"/>
    <ss access="16#c093#" base="16#0000#" limit="16#ffff_ffff#" selector="16#0010#"/>
    <tr access="16#008b#" base="16#0000#" limit="16#ffff#" selector="16#0018#"/>
    <ldtr access="16#0001_0000#" base="16#0000#" limit="16#0000#" selector="16#0000#"/>
   </segments>
  </registers>
 </vcpu>
```

Listing 11.2: VM vCPU

Chapter 12

Bibliography

[1] Adrian-Ken Rueegsegger and Reto Buerki. *Muen System Specification.*