User's Guide

QNX Software Systems Limited
1001 Farrar Road
Ottawa, Ontario
K2K 0B3
Canada

Voice: +1 613 591-0931
Fax: +1 613 591-3579
Email: info@qnx.com
Web: http://www.qnx.com/

Electronic edition published: March 06, 2020

Contents

About This Document

The *QNX SMMUMAN and QNX SMMUMAN for Safety User's Guide* describes how to install and use the QNX SMMUMAN, and QNX SMMUMAN for Safety. Throughout this document when we refer to SMMUMAN, the `smmuman` service, etc., unless we explicitly state otherwise, we are referring to both safety and non-safety variants.

DANGER:

If you are building a safety-related system, you *must* use the QNX SMMUMAN for Safety variant which has been built and approved for use in the type of system you are building, and you must use the SMMUMAN only as specified in its *Safety Manual*.

SMMUMAN components that have been certified for a safety-related system have the suffix "-safety" (e.g., `foo`**`-safety`**`.so`). Only these safety components may be used in a safety-certified system.

What's in this guide

The following table may help you find information in this guide quickly:

To find out about:	See:
DMA device containment; and SMMUMAN and its components	"*Architecture*"
Installing and starting the `smmuman` service	"*Installation*" and "*Starting and stopping smmuman*"
The `smmuman` service and its responsibilities	"*Mapping DMA devices and memory regions*"
Configuring the `smmuman` service	"*Configuring smmuman*"
Using the `smmuman` service client API	"*Mapping DMA devices and memory regions*" and "*SMMUMAN Client API Reference*"

Typographical conventions

Throughout this manual, we use certain typographical conventions to distinguish technical terms. In general, the conventions we use conform to those found in IEEE POSIX publications.

The following table summarizes our conventions:

Reference	Example
Code examples	`if( stream == NULL)`
Command options	`-lR`
Commands	`make`
Constants	NULL
Data types	`unsigned short`
Environment variables	*PATH*
File and pathnames	**/dev/null**
Function names	*exit()*
Keyboard chords	**Ctrl–Alt–Delete**
Keyboard input	`Username`
Keyboard keys	**Enter**
Program output	`login:`
Variable names	*stdin*
Parameters	*parm1*
User-interface components	**Navigator**
Window title	**Options**

We use an arrow in directions for accessing menu items, like this:

You'll find the Other... menu item under **Perspective** → **Show View**.

We use notes, cautions, and warnings to highlight important messages:

Notes point out something important or useful.

CAUTION: Cautions tell you about commands or procedures that may have unwanted or undesirable side effects.

DANGER: Warnings tell you about commands or procedures that could be dangerous to your files, your hardware, or even yourself.

Note to Windows users

In our documentation, we typically use a forward slash (/) as a delimiter in pathnames, including those pointing to Windows files. We also generally follow POSIX/UNIX filesystem conventions.

Technical support

Technical assistance is available for all supported products.

To obtain technical support for any QNX product, visit the Support area on our website (*www.qnx.com*). You'll find a wide range of support options, including community forums.

Chapter 1
Architecture

This chapter presents the architecture of the SMMUMAN, and its responsibilities and behavior.

The QNX System Memory Management Unit Manager (SMMUMAN) is a system memory management unit (IOMMU/SMMU) manager that runs the following board architectures: ARM and x86, and makes use of the DMA containment and memory-management support available for these architectures.

On ARM platforms, IOMMU/SMMU components are usually called “System Memory Management Units” (SMMUs); on Intel x86 platforms, this technology is usually called “Virtualization Technology for Directed I/O” (VT-d).

In this document we use “IOMMU/SMMU” to refer to the component on any supported hardware platform, unless referring to a component for a specific architecture or board, exclusive of other architectures or boards, in which case we use the architecture-specific or board-specific acronym (e.g., “VT-d” (Intel x86), “SMMU” (ARM), “IPMMU” (ARM: Renesas R-Car boards)).

Supported board architectures and required hardware

The SMMUMAN can run on ARM or x86 platforms. If it is running in a guest in a QNX Hypervisor VM, that VM must be configured to present the functional equivalent of the underlying hardware platform to its guest.

ARM

To support the SMMUMAN, ARM platforms require the following:

- System Memory Management Unit (SMMU), or equivalent (e.g., IPMMU on Renesas R-Car boards)

x86

To support the SMMUMAN, x86 platforms require the following:

- Virtualization Technology for Directed I/O (VT-d)

Supported OSs

The SMMUMAN can be included in:

- QNX Neutrino OS
- QNX OS for Safety (QOS)

It is intended for use on the supported hardware platforms with the following software:

- the QNX Neutrino OS kernel running directly on hardware
- the QNX Neutrino OS kernel with the virtualization extension, running directly on hardware as a hypervisor host (QNX Hypervisor)
- the QNX Neutrino OS kernel running as a guest in a QNX Hypervisor VM
- the QNX Neutrino kernel for safety (QOS), running directly on hardware

- the QOS with the virtualization extension, running directly on hardware as a hypervisor host (QNX Hypervisor for Safety (QHS))
- the QNX Neutrino OS kernel running as a guest in QHS VM
- the QOS running as a guest in a QHS VM

Design Safe State (DSS)

When the SMMUMAN or any of its components meets an unknown or undefined condition, it attempts to enter its Design Safe State (DSS). This DSS is to exit.

DANGER: If you are using SMMUMAN for Safety, see the Safety Manual for more information about the DSS.

Isolation of DMA devices

The SMMUMAN supports spatial isolation of DMA devices on systems with IOMMU/SMMUs.

Isolation

The figure below presents the four isolation axes that can be implemented in a software system. Spatial isolation is fundamental to all other forms of isolation.

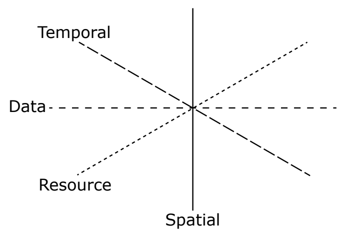

Figure 1: The isolation axes: spatial, resource, data, temporal.

In a QNX system, the OS kernel (`procnto`) uses MMU page tables to control attempts to access memory. In a virtualized system with the QNX Neutrino OS plus its virtualization extension (a `libmod_qvm` variant) running as the host, a second layer of page tables (or intermediate page tables) is used to manage guest access to memory.

MMUs can’t be used to manage Direct Memory Access (DMA) device attempts to access memory, however. The following section describes a different hardware mechanism, called an IOMMU/SMMU, which manages DMA device access to memory.

On ARM platforms, intermediate page tables are known as Stage 2 page tables; on x86 platforms they are known as Extended Page Tables (EPT)).

DMA devices and IOMMU/SMMUs

A non-CPU initiated read or write is a read or write request from a DMA device (e.g., GPU, network card, sound card).

The CPU doesn't control memory access by a DMA device. Instead, a DMA device takes control of the memory bus to gain direct access to system memory. Since no CPU is implicated in the memory access, an OS can’t manage a DMA device’s access to system memory without support from hardware IOMMU/SMMUs.

Most importantly, the protections the OS can provide against incorrect (and possibly malicious) access to memory by requests that go through the CPU do not apply to access requests from DMA devices. The OS cannot ensure that a DMA device is prevented from accessing memory it is not authorized to access.

A System Memory Management Unit (IOMMU/SMMU) is a hardware component that provides translation and access control for non-CPU initiated reads and writes, similar to the translation and access control page tables provide for CPU-initiated reads and writes.

Hardware limitations

Note that on some boards the IOMMU/SMMU hardware doesn't provide the `smmuman` service the information it requires to map individual devices and report their attempted transgressions.

For example, on some ARM boards the IOMMU doesn't provide the `smmuman` service the information it requires to identify individual PCI devices. Similarly, on x86 boards the VT-d hardware can't identify or control individual MMIO devices that do DMA. Additionally, some boards might not have enough session identifiers (SIDs) to be able to assign a unique SID to every hardware device, so multiple hardware devices may have to share an SID.

Pass-through DMA devices in virtualized systems

In a virtualized system, pass-through devices are devices that are "owned" by a guest running in a VM. A driver in the guest controls the device hardware directly.

For a DMA device to be usable as a pass-through device in a virtualized system, an IOMMU/SMMU is required for the following reasons:

- A DMA device that is passed through to a guest won't work if its memory access is restricted to the host-physical memory regions assigned for the guest's memory. It requires its own region in host-physical memory, and this region must be mapped to guest-physical memory.

  An IOMMU/SMMU is required to map guest-physical addresses visible to the DMA device to host-physical addresses. Since the DMA device is owned by the guest, it is configured to output guest-physical addresses to the bus, and the IOMMU/SMMU is needed to convert these addresses to host-physical addresses before they are passed on to the memory controller.

- The hypervisor host layer has no knowledge of a device that is passed through to a guest, and a DMA device's memory access doesn't go through a CPU, which could trap transgressions. If the guest OS fails to notice that a DMA device is misbehaving, in the absence of an IOMMU/SMMU, no further checks are available.

  To protect against misbehaving pass-through DMA devices, an IOMMU/SMMU must be programmed with the memory regions that each DMA device (regardless of ownership) is permitted to access.

SMMUMAN components

This section describes the SMMUMAN components.

The SMMUMAN comprises a service, an API library, and support libraries (or drivers):

smmuman

The architecture-agnostic SMMUMAN service itself; it is a resource manager that provides services to SMMUMAN clients through the `libsmmu` API library (see below, and "*The libsmmu.a client-side API*").

smmu-*.so

Architecture-specific and board-specific libraries; these provide the interface between the `smmuman` service's architecture-agnostic code and the hardware IOMMU/SMMUs.

libsmmu.a

The API that SMMUMAN clients use to access the SMMUMAN services (see "*The libsmmu.a client-side API*").

WARNING:

Only SMMUMAN safety components may be used in a systems that require safety certification.

There is only one variant of the **libsmmu.a** library. This variant is the safety variant, and may be used in systems that require safety certification. All other SMMUMAN components have two variants: standard and safety. The safety components have the suffix "-safety" (e.g., smmuman-**safety**).

Preferentially, the SMMUMAN safety variant (`smmuman-safety`) loads the safety variants of support files. For example, for NXP i.MX8 platforms, it loads **smmu-armsmmu-safety.so** and **smmu-cfg-imx8-safety.so** (see "*Configuration at startup*" in the "Configuring `smmuman`" chapter).

The figure below presents a high-level view of the SMMUMAN components. For the purposes of this illustration, we have used the components for the x86 boards, whose SMMUs are called VT-ds. The architecture-specific SMMUMAN support library is **smmu-vtd.so**.

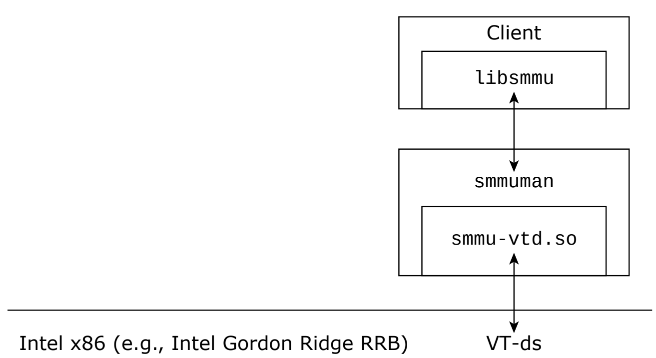

Figure 2: A high-level overview of the SMMUMAN.

The **smmuman** | **smmuman-safety** service

The smmuman service is architecture-agnostic. It requires architecture-specific or board-specific libraries to interface with board IOMMU/SMMU. The smmuman service looks after the following:

- Loading and parsing the user-input configuration information at startup.
- Replacing the board configuration information with user-input configuration information, where relevant.
- Optionally, using this information to inform the board IOMMU/SMMU units of the DMA devices on the system, and of the permitted memory ranges for each device, as well as the activity permitted for each of these memory ranges and DMA devices (read-only, read-write).
- In response to client requests, programming the IOMMU/SMMUs on the board (see "*Mapping DMA devices and memory regions*").
- Monitoring the IOMMU/SMMUs on the board and recording illegal DMA devices attempts to access memory that have been communicated by the IOMMU/SMMU unit in conjuction with the support code (see "*The smmu-*.so libraries*").

The **smmu-*.so** libraries

The **smmu-*.so** libraries are architecture-specific and board-specific libraries used by the smmuman service to interface with board IOMMU/SMMU units. These libraries look after the following:

- Implement the architecture-specific and board-specific functions the smmuman service needs to be able to communicate with the board IOMMU/SMMU units, including the retrieval from the board firmware configuration information about the presence and locations of DMA devices.

The SMMUMAN includes the following architecture-specific and board-specific support libraries:

smmu-armsmmu.so | smmu-armsmmu-safety.so

Implement the code to communicate with ARM SMMUs as specifed in *ARM System Memory Management Unit Architecture Specification: SMMU architecture version 2.0* (2016) ARM IHI 0062D.c (ID070116). The SMMUMAN uses this library on boards such as the NXP i.MX8.

To support configurable StreamIDs on NXP i.MX8 platforms, the SMMUMAN also provides the **smmu-cfg-imx8.so** | **smmu-cfg-imx8-safety.so** libraries; these libraries set the StreamIDs according to the configuration specified in the smmuman service configuration (see "*Board-specific configuration libraries*").

smmu-rcar3.so | smmu-rcar3-safety.so

Implement the code to communicate with Renesas R-Car H3 IPMMUs, as specified in Chapter 16 of *Renesas R-Car Series, 3rd Generation User's Manual: Hardware*, Nov. 2018 (Rev. 1.50).

smmu-vtd.so | smmu-vtd-safety.so

Implement the code to communicate with Intel x86 VT-ds, as specified in *Intel Virtualization Technology for Directed I/O Architecture Specification*, Nov. 2017 (D51397-009, Rev. 2.5).

If you are using the safety variant of this support library (**smmu-vtd-safety.so**), you must include the **pci_server-qvm_support.so** in your system (see "*Safety variant support for PCI (x86)*" in the "smmuman" chapter).

SMMUMAN in a guest OS running in a QNX Hypervisor VM uses the vdev-smmu virtual device, and doesn't require a **smmu-*.so** library (see "*SMMUMAN in a QNX Hypervisor guest*").

If you need to use the SMMUMAN on another board, you will need an appropriate support library. For more information, contact your *QNX representative*.

The libsmmu.a client-side API

Processes that need to use the SMMUMAN services can use the API presented in the libsmmu library. For a description of the API, see "*SMMUMAN Client API Reference*"; for instructions on how a SMMUMAN client should use it, see "*Mapping DMA devices and memory regions*."

SMMUMAN in a QNX Hypervisor guest

SMMUMAN can be used in a QNX guest running in a QNX Hypervisor VM.

No `smmu-*.so` library is required for a SMMUMAN running in a guest in a QNX Hypervisor VM, as shown in the figure below. The `vdev-smmu` virtual device in the VM provides the required subset of the **smmu-*.so** library functionality needed by a SMMUMAN running in a guest (see "*The vdev-smmu virtual device*").

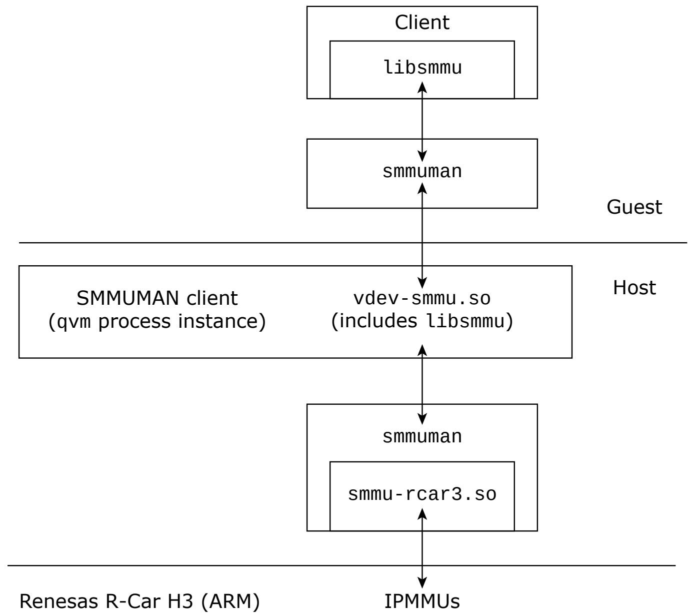

Figure 3: A high-level overview of the SMMUMAN in a guest running in a QNX Hypervisor VM.

The figure above shows the SMMUMAN in a QNX guest running in a QNX Hypervisor VM, as well as the SMMUMAN running in the hypervisor host. Note that as far as the SMMUMAN in the host is concerned the `qvm` process instance is simply a client like any other.

The `vdev-smmu` virtual device

To support the SMMUMAN running in a QNX guest in a QNX Hypervisor VM or QNX Hypervisor for Safety VM, the VM hosting the guest must include the `vdev-smmu` virtual device (vdev). This vdev provides the following:

IOMMU/SMMU services

The `vdev-smmu` vdev provides for the guest the same services for guest-physical memory as an IOMMU/SMMU provides for an OS running on the hardware: accept configuration information passed to it by the guest's `smmuman` service, deny DMA device attempts to access memory outside their permitted regions, communicate these denials back to the devices, record these attempts, etc.

Memory mapping

The `vdev-smmu` vdev shares memory mappings for DMA devices passed-through to the guest with the SMMUMAN running in the hypervisor host to ensure that the host SMMUMAN and the board IOMMU/SMMUs are able to manage pass-through DMA devices' accesses to host-physical memory.

Supporting components

The **vdev-smmu.so** virtual device shared object is a QNX Hypervisor component. It must be loaded into the hypervisor host to enable the `smmuman` service in a hypervisor guest. For more information about `vdev-smmu` and how to use it, see "*SMMUMAN in a QNX Hypervisor system*" in the "*Mapping DMA devices and memory regions*" chapter, and the *QNX Hypervisor User's Guide*.

To run the `smmuman` service, a guest running in a QNX Hypervisor VM on ARM platforms must load **libfdt.so**. Make sure you include this shared object in the guest's buildfile.

Chapter 2
smmuman

System memory management unit (IOMMU/SMMU) manager

Syntax:

```
smmuman options
```

Runs on:

QNX Neutrino

Options:

Options may be specified in the command line, or in a ***.smmu** configuration file, or both.

The smmuman service reads its startup configuration input *in a single pass*, from start to finish. If an option is specified more than once, the smmuman service does one of the following:

- If more than one instance of an option will result in a usable configuration, the smmuman service adds each option instance to the configuration, so that the resulting configuration will have multiple instances of the information supplied by each option instance.

  For example, multiple instances of the allow option can be used to allow multiple DMA devices access to different memory regions, so the smmuman service will use every instance of this option it encounters.

- If only a single instance of an option will have any effect on the configuration, or if multiple instances will result in an unusable configuration, the smmuman service uses the last instance of the option it encounters.

  For example, only one instance of the *foreground* option is usable, since smmuman runs either in the foreground or in the background.

For more information, see "*Configuring smmuman*," "*Global options*," and the architecture-specific and board-specific entries in this chapter.

Description:

In all implementations, the smmuman service:

- Enables device drivers and user applications (smmuman clients) to determine the memory regions DMA devices are permitted to access.
- Programs the IOMMU/SMMUs with the memory regions each DMA device is permitted to access, as determined by the configuration and/or requested by the smmuman clients.
- Ensures that, after DMA devices' memory access permissions have been programmed into the IOMMU/SMMUs, no DMA device is able to access memory to which it has not been explicitly granted access.
- Enables the smmuman clients to retrieve records of DMA device attempts to access memory that have been rejected by the IOMMU/SMMUs.

- Manages reserved memory regions, adding them to allowed mappings when a DMA device is enabled, and removing them when the device is disabled.

CAUTION:

Until the `smmuman` service programs the IOMMU/SMMUs on a board with a DMA device's memory access permissions, that device has unrestricted access to memory.

Programming the IOMMU/SMMUs with DMA devices' memory access permissions specifies both the memory regions these devices may access and their permissions when accessing these regions (e.g., read only, read and write). It doesn't affect memory access managed by board MMUs (i.e., access that passes through the CPU).

When implemented in the host of a QNX Hypervisor system, in addition to the responsibilities listed above, the SMMUMAN:

- Manages guest-physical memory to host-physical memory translations and access for non-CPU initiated reads and writes (i.e., for DMA devices).
- Ensures that no pass-through device can access host-physical memory outside its mapped (permitted) host-physical memory region.

Installation

This section describes how to get SMMUMAN or SMMUMAN for Safety and add it to a QNX Neutrino OS, QNX for Safety (QOS), or other QNX system.

SMMUMAN for Safety

Two variants of SMMUMAN are available:

- SMMUMAN
- SMMUMAN for Safety

If you are building a safety-related system (i.e., you are using QNX for Safety (QOS), QNX Platform for Instrument Clusters (QPIC), QNX Hypervisor for Safety (QHS), or any other QNX for Safety product) remember that *the functional safety of your system depends on the correct implementation of SMMUMAN*.

This means that:

- You *must* use SMMUMAN for Safety.
- You must use SMMUMAN for Safety in accordance with the requirements and restrictions stated in the QNX SMMUMAN for Safety *Safety Manual*.

WARNING: SMMUMAN for Safety components include the "-safety" suffix. If you are building a safety-related system, do *not* use any SMMUMAN components that don't have this suffix.

Assumptions

These instructions assume that:

- You have the appropriate development environment installed and configured on your development host (see the QNX SDP 7.*n* and QOS 2.*n* documentation).
- You know how to build a QNX Neutrino system (see *Building Embedded Systems* in the QNX SDP 7.*n* documentation).
- You know how to use the QNX Software Center (see the *myQNX License Manager and QNX Software Center User's Guide*).

WARNING: If you are building a safety-related system, don't use macOS as a development host for your safety system. Only the Windows and Linux development hosts identified in the QOS 2.*n* documentation are qualified for ISO26262-based safety system development. Although the QNX Software Development Platform supports macOS as a development host, the tool chain on macOS isn't qualified for ISO26262-based safety system development.

Adding SMMUMAN to a system

To implement SMMUMAN on your system, assuming that all the SMMUMAN components were included in your QNX SDP package, in your development environment:

1. Modify your BSP buildfile to include the SMMUMAN components:

- **smmuman.so** or **smmuman-safety.so**, as required
- for a system running on the hardware (i.e, a hypervisor host) the architecture-specific or board-specific library for your board: **smmu-*** or **smmu-*-safety**, as required (see "*The smmu-*.so libraries*"), and the ***.smmu** configuration file for the board, as required (see "*Configuring smmuman*")
- for a system that will be running the `smmuman` service safety variant (**smmuman-safety.so**) on an x86 platform, the **pci_server-qvm_support.so** support shared object (see "*Safety variant support for PCI (x86)*" below)
- for a QNX Hypervisor or QNX Hypervisor for Safety (QHS) system, the **vdev-smmu.so** IOMMU/SMMU virtual device shared library, so that it will be available for guests that need to use the `smmuman` service

2. For a system running as a QNX guest in a QNX Hypervisor or QHS VM, in addition to adding the SMMUMAN components in your guest build, add the `vdev smmu` option in the configuration for the hypervisor VM that will host the guest (see "*The vdev-smmu virtual device*").
3. Run `make` to rebuild your system with the SMMUMAN components.

Example of entries to add to the buildfile

Below are some examples of how the buildfile might be modified to include SMMUMAN.

x86 for an ordinary system

For a non-safety system running on a supported x86 board, the buildfile might include the following:

```
/bin/smmuman = smmuman
/lib/dll/smmu-vtd.so = smmu-vtd.so
```

where `smmuman` is the SMMUMAN service, and `smmu-vtd.so` is the SMMUMAN support library for x86 platforms.

ARM Renesas R-Car H3 for a safety-related system

For a safety-related system running directly on a supported Renesas R-Car H3 board, the buildfile might include the following:

```
/bin/smmuman-safety = smmuman-safety
/bin/smmu-rcar3-safety.so = smmu-rcar3-safety.so
/etc/smmuman/rcar-h3-safety.smmu = ./smmuman-config/rcar-h3-safety.smmu
```

where `smmuman-safety` is the SMMUMAN for Safety service, `smmu-rcar-safety.so` is the SMMUMAN support library, and **rcar-h3-safety.smmu** is your safety `smmuman` configuration file for Renesas R-Car H3 platforms.

Guest in QNX Hypervisor for Safety

For a safety-related system running as a guest in a QNX Hypervisor for Safety VM, the guest's buildfile might include the following:

```
/bin/smmuman-safety = smmuman-safety
```


where `smmuman-safety` is the SMMUMAN for Safety service, and `vdev-smmu-safety` is the QNX Hypervisor virtual IOMMU/SMMU device in the hosting VM.

ARM Renesas R-Car H3

For a system running on a supported Renesas R-Car H3 board, the buildfile might include the following:

```
/bin/smmuman = smmuman
/bin/smmu-rcar3.so = smmu-rcar3.so
/etc/smmuman/rcar-h3.smmu = ./smmuman-config/rcar-h3.smmu
```

where `smmuman` is the `smmuman` service, `smmu-rcar.so` is the SMMUMAN support library, and **rcar-h3.smmu** is your `smmuman` configuration file for Renesas R-Car H3 platforms.

QNX guest in QNX Hypervisor system

For a system running as a guest in a QNX Hypervisor VM, the guest's buildfile might include the following:

```
/bin/smmuman = smmuman
```

where `smmuman` is the `smmuman` service.

The host buildfile will have to include the **vdev-smmu.so** virtual device, and the configuration for the VM hosting the guest will have to include the `vdev smmu` option.

Safety variant support for PCI (x86)

If you are using the safety variant of the VTD support library (**smmu-vtd-safety.so**), you must include the **pci_server-qvm_support.so** support file in your system.

To load this file, add the configuration information to one of:

- an in-line file in your buildfile
- a separate configuration file

In-line file in buildfile

You can use your buildfile to configure your startup to load the **pci_server-qvm_support.so** support file. For example, you can add the following in-line file to your buildfile:

```
pci_server.cfg = {
 [runtime]
 PCI_SERVER_MODULE_LIST=pci_server-qvm_support.so
}
```

which, because no paths are specified, will use the default locations: **pci_server.cfg** in **/proc/boot**, and **pci_server-qvm_support.so** in **/lib/dll/pci**, the directory where `pci-server` expects to find all its ***.so** files.

Separate configuration file

You can add the configuration information to a separate file and point `pci-server` to it at startup:

```
pci-server --config=pathto/pci_server.cfg
```

where *pathto* is the path to the directory with your configuration file.

Whichever method you use, if you put the server module in its default location (**/lib/dll/pci**), you don't need to specify the full path to the module in your configuration. If you place the server module in another location, specify the full path.

After your system has booted, you can use `pidin` or `slog2info` to confirm that the server module has loaded; for example:

```
pidin -ppci-server libs
```

For more information about including configuration information in your buildfile, see "OS Image Buildfiles" in *Building Embedded Systems*. For more information about configuring `pci-server`, see "`pci-server`" in the QNX Neutrino *Utilities Reference*.

Starting and stopping smmuman

Typically, you start the smmuman service by including the instructions in your system startup script.

When the smmuman service is included in the bootable OS image, it can be started as part of the system startup procedure. For example, the following snippets are for a buildfile that includes smmuman in a build for a hypervisor host on a fictional x86 board ("myx86"), and starts the service:

```
#
[image=0x2000000]
[virtual=x86_64,kpi +compress] boot = {
    startup-intel-MX86 -D8250_mmio.0xFC000000^2.115200 -v
[+keeplinked module=aps module=qvm] \
 PATH=/sbin:/bin:/usr/sbin:/usr/bin:/opt/sbin:/opt/bin:/proc/boot \
 LD_LIBRARY_PATH=/lib:/usr/lib:/lib/dll:/lib/dll/pci:/proc/boot procnto-smp-instr \
}

[+script] startup-script = {
    display_msg Welcome to QNX Neutrino 7.0 x86_64 on my x86 board.

...

#Concatenate above, the BSP build file of your choosing

[+script] myx86-startup-script = {

    ...

    #Start smmuman and point it to the configuration file.
    smmuman @/etc/myx86.smmu
}

# My board binaries #

[data=c]

[perms=0444] /root/envset.sh {
export PCI_HW_MODULE=/lib/dll/pci/pci_hw-Intel_x86_MX86.so
}

...

#Include smmuman configuration file in build.
/etc/myx86.smmu = ${MY_TARGET}/etc/myx86.smmu

...
```

```
#Include smmuman in build.
/bin/smmuman = smmuman
/lib/dll/smmu-vtd.so = smmu-vtd.so
```

CAUTION: You should start the `smmuman` service before you start any DMA-enabled drivers.

Determining where the smmuman service is running

A `smmuman` service may run at the host layer or as part of a guest in a QNX Hypervisor VM. When it runs at the host layer, the service may need to load a support library. If it is running in a guest in a VM, it needs the `vdev-smmu` virtual device.

When the `smmuman` service processes its configuration information, it looks for the `vdev-smmu` virtual device:

- If it finds this vdev, it knows that it is running as a guest in a VM and proceeds with its startup.
- If it doesn't find this device, it assumes that it is running in the host, and attempts to load the **smmu-*.so** support library specified by the `smmu` option in its configuration before proceeding with its startup.

If the `smmuman` service doesn't find the `vdev-smmu` virtual device and is unable to load the specified support library or none is specified, it reports that the required hardware isn't present and moves to its DSS (see "*Design Safe State (DSS)*" in the "*Architecture*" chapter).

Startup in a QNX Hypervisor guest

To use the `smmuman` service in a QNX guest running in a QNX Hypervisor VM, start it as you would the `smmuman` service for a system running directly on the hardware (see above). When the `smmuman` service is implemented in a guest in a QNX Hypervisor VM it proceeds with startup as described above, with the following differences:

1. The `smmuman` service doesn't load a support library, but communicates directly with the IOMMU/SMMU virtual device (`vdev-smmu` vdev).

   If it doesn't find a `vdev-smmu` in its hosting VM (`qvm` process instance), the `smmuman` service in the guest reports that the required hardware isn't present and moves to its DSS.

2. When it parses its configuration information, the `smmuman` service ignores any `reserved`, `smmu`, `unit`, or `use` options it finds in its configuration, because these options are relevant only to the corresponding `smmuman` service running in the hypervisor host.
3. The `smmuman` service queries the `vdev-smmu` vdev, which confirms that a `smmuman` service is running in the QNX Hypervisor host and that the hosting VM is attached as a client of the host-level `smmuman` service, and provides the safety-certification status of all the components on which the guest's `smmuman` service relies:

   - the host-level microkernel and process manager (`procnto*`)
   - the host-level SMMUMAN components (**smmuman**, **libsmmu.a**, and the **smmu-*.so** support library)
   - the hypervisor components (`qvm` and the **vdev-*.so** virtual devices)

The **vdev-smmu.so** virtual device shared object is a QNX Hypervisor component. For more information about this vdev and how to use it, see the QNX Hypervisor documentation.

Component safety-certification status

By default, if the `smmuman` service running in the guest is a `smmuman` for safety variant (`smmuman-safety`) all the SMMUMAN and all the QNX Hypervisor components must be safety-certified variants. If the required components are not safety-certified variants, the `smmuman` service in the guest moves to its DSS.

The behavior described above is the *default* behavior for `smmuman-safety`. This behavior may be modified with the `safety` option (see *`safety`* in "*Global options*").

Stopping the `smmuman` service

WARNING:

You should never stop the `smmuman` service after it has started. If you have implemented SMMUMAN in your system, the integrity of your system depends on the `smmuman` service running continuously.

If for whatever reason the `smmuman` service moves to its DSS (see "*Design Safe State (DSS)*"), your system should move to its DSS.

Configuring smmuman

Correct configuration of the smmuman service requires careful attention to architecture-specific and board-specific details provided in the board manufacturer’s documentation.

The smmuman configuration:

- specifies how the smmuman service should run
- describes the IOMMU/SMMU devices available on the board, if these are not available from the board firmware (required for ARM architectures)
- optionally, defines the memory regions that DMA devices are permitted to access

The smmuman service programs the configuration information about DMA devices, and memory regions and permissions into the hardware IOMMU/SMMUs. The IOMMU/SMMUs (or vdev-smmu) use the configuration to limit DMA devices' memory access to their permitted regions, and for their configured permissions.

If the smmuman service is running in a guest in a QNX Hypervisor VM (qvm process instance), the service passes the configuration information for the guest's pass-through DMA devices to the vdev-smmu virtual device. This virtual device then passes this information on to the smmuman service in the hypervisor host to configure the IOMMU/SMMUs on the board, ensuring that devices passed through to a guest are restricted to their permitted memory regions and permissions in host-physical memory.

You don't have to include the smmuman configuration file in your build file, or have the service load it with your system startup. You can, for instance, store the smmuman configuration file on a secondary storage device, to be loaded after the primary IFS is up and running. This might be a useful strategy if your system uses two IFSs, and the primary IFS doesn't include DMA devices.

Configuration at startup

The smmuman service needs a minimum set of configuration information at startup. This set includes information about:

- how the service should run (see “*Configuring how the smmuman service runs*”)
- the hardware on the board (see “*Describing the hardware*”)

At startup, the smmuman service reads its configuration input *in a single pass*, from start to finish. The service usually receives this input through some combination of information:

- entered through the command line
- entered through one or more configuration files
- retrieved from the board firmware
- supplied by the architecture-specific and board-specific support libraries

Configuration information precedence

You may have the `smmuman` service retrieve information from the board at startup, then add user-input information to it. By default, user-input information will override the information obtained from the board.

For information about how the `smmuman` service handles multiple instances of the same option, see "*Options*" in this chapter.

The information supplied at startup may in some cases also include memory mappings and DMA device permissions for these mapped memory regions. Note, however, that the recommended method for configuring memory mappings and device permissions is through the SMMUMAN client API (see "*Mapping DMA devices and memory regions*" in this chapter).

Default and preferred support files

The non-safety variant of the SMMUMAN service (`smmuman`) only attempts to load the the non-safety variants of support files. For example, if you start the non-safety variant of the `smmuman` service thus: `smmu vtd`, the service will always try to load only the non-safety support file: **smmu-vtd.so**.

However, the service's *safety* variant (`smmuman-safety`) preferentially loads the safety variants of support files. If it doesn't find the safety variant of the support file, it then attempts to load the non-safety variant. For example, if you start the `smmuman` service's safety variant thus: `smmu vtd`, the service will try to load **smmu-vtd-safety.so**; if this fails, it will then try to load the non-safety support file: **smmu-vtd.so**.

You can force either variant of the `smmuman` service to load only the specified variant of a support file by naming the support file. For example, assuming that you configured the service to move to its DSS if a required safety component isn't present, if you start the service thus: `smmu vtd-safety.so`, it will attempt to load only the named support file, and move to its DSS if it doesn't find the file.

Independently of its startup instructions, the `smmuman` service checks the safety-certification status of the components on which it depends. You can use the `safety` option to configure service's response to the presence or absence of safety-certified components on which it relies (see the *`safety`* option in this chapter).

Startup configuration syntax

In the user-input configuration (file or command line), anything that follows a "#" character is considered a comment and is ignored. An "@" character indicates that what follows is a file name.

Configuration files may be identified by their "**.smmu**" suffix (e.g., **mysystem.smmu**).

Thus, in the startup instructions, to use a configuration file, enter `smmuman`, then an `@` followed by the filename. For example:

```
smmuman @mysystem.smmu
```

In a production system, you should start the `smmuman` service as part of the system startup, with a configuration file specified in the startup script. For example:

```
smmuman @/etc/foo.smmu
```

This character can be used to nest configuration files. For example, if a configuration file **foo.smmu** includes a line: `@moo.smmu`, then when it encounters this line the `smmuman` service will read its configuration information from **moo.smmu** before continuing to read **foo.smmu**.

CAUTION: An error in the command-line input can produce unexpected results. Use the command line only to supplement the information in the configuration files during development and for troubleshooting.

Textual substitutions

As it reads through its configuration information, `smmuman` performs textual substitutions to its configuration information when it encounters the following character sequences:

$env{*envvar*}

Replace the entire text string above with the contents of the *envvar* environment variable as found in the system page.

$asinfo_start{*asinfo_name*}

Replace the entire text string above with the start address of the system page *asinfo* entry specified by *asinfo_name* in the system page.

$asinfo_length{*asinfo_name*}

Replace the entire text string above with the length of the system page *asinfo* entry specified by *asinfo_name* in the system page.

You can use this textual substitution to make your configuration more robust by having the host startup place values in the system page *asinfo* entry where they can be picked up by `smmuman`.

For example, below is a partial `smmuman` configuration for an x86 board that picks up memory allocations from the system page:

```
debug 2

smmu vtd require

allow smbs0   $asinfo_start{smbs0},$asinfo_length{smbs0},st
allow xhci    $asinfo_start{xhci},$asinfo_length{xhci},st
...

device pci:0:18.0 smbs0
device pci:0:21.0 xhci
...
```

Note that when using `$asinfo_start`, pass the leaf name only, and *not* the full path. For example, the following is incorrect:

```
allow ahci $asinfo_start{/memory/below4G/ram/testMem},10M,st
```

but the following is **correct**:

```
allow ahci $asinfo_start{testMem},10M,st
```

For more information about the system page *asinfo* data structure array, see the “System Page” chapter in *Building Embedded Systems*.

Configuring how the smmuman service runs

The `smmuman` service supports options that determine how it will run: (e.g., in the foreground, outputting debug information). These options must be specified at startup. For more information, see “*Global options*” in this chapter.

Describing the hardware

When it starts, the `smmuman` service needs to know about the IOMMU/SMMUs and the DMA devices on the board. The information it requires includes:

- the architecture-specific and board-specific support library or libraries to load, or in their absence in the configuration, confirmation that the service is running in a QNX Hypervisor virtual machine (see “*Determining where the smmuman service is running*”)
- the type(s) of IOMMU/SMMUs on the board, and their locations, and where to look for any additional information needed in order to program the IOMMU/SMMUs
- descriptions of the DMA devices available on the board
- information about which IOMMU/SMMU controls which DMA devices

Sources of hardware descriptions

If `smmuman` is running on the hardware (i.e., not in a QNX hypervisor VM) it obtains architecture-specific and board-specific information from the **smmu-*.so** support libraries. Additional information may be available in different locations on the hardware, depending on the board architecture and the specific board itself. To obtain some of this information, the `smmuman` service with its support libraries can query the board.

For x86 boards, the hardware description should be available in the board’s ACPI tables. A configuration file is required only for non-standard implementations. For ARM boards, some information may be available in board-specific code. For both architectures, information that isn't provided by the board, must be provided in the `smmuman` configuration.

Source of virtual hardware descriptions

If `smmuman` is running in a QNX Hypervisor VM in a hypervisor system, no support libraries are required; the information for the virtual hardware is provided by the `vdev-smmu*` virtual IOMMU/SMMU device.

Without proper guidance from the system designer the `smmuman` service can't know where the information it needs resides on the board. On x86 platforms, the ACPI tables are usually at a standard location; on ARM platforms the location or locations of the information the SMMUMAN needs is less predictable.

Always consult your board manufacturer's documentation for the locations and values you must use for your SoC. These locations and values may change, *even between board revisions*.

Mapping devices and memory regions

The `smmuman` service includes options that can be used in the startup configuration to specify memory mappings and DMA device permissions for these mapped memory regions. These mappings and permissions are not mandatory; the service doesn't need them to complete its startup successfully.

In fact, the recommended method for configuring memory mappings and device permissions is not through the startup configuration, but through the SMMUMAN client API in **libsmmu.a** (see "*Mapping DMA devices and memory regions*" in this chapter, and the "*SMMUMAN Client API Reference*" chapter).

Global options

Options for the `smmuman` service may be specified in the command line, or in a configuration file (e.g., **mysystem.smmu**), or both.

The `smmuman` configuration uses *options*, which may have *arguments*. Options and arguments available for all supported hardware platforms are described below.

About notation

The default notations (no prefix needed) for specifying memory addresses and sizes are:

- address in memory – hexadecimal
- size or length of memory region – decimal

For example, `allow foo 4000,4096` refers to a memory region that starts at address `0x4000` and has a size of 4096 (`0x1000`) bytes .

If you prefer to write a memory address or region size with a notation other than the default, you can use a prefix to specify the notation:

- decimal – `0d` (e.g., `0d1234`)
- hexadecimal – `0x` (e.g., `0x4D2`)

Thus, the following are equivalent:

```
allow foo 0x4000,4096
allow foo 4000,4096
allow foo 0x4000,0x1000
allow foo 0x4000,0d4096
allow foo 0d16384,4096
```

```
allow foo 0d16384,0d4096
allow foo 0d16384,0x1000
```

You can use size multipliers: “K”, “M”, “G” (or “k”, “m”, “g”) in the address and length arguments; for example: `allow foo 4K,1k` is equivalent to `allow foo 0x1000,0x400`. (Remember: the size multipliers are *decimal* multipliers, so `4K` is `4 x 1024 = 4096`, or `0x1000`.)

Other numeric configuration values are specified in decimal; for example, in `device pci:0:18.0 foo` the values for *bus*, *dev*, and *func* are decimal values (see *device* below).

> We recommend that, to avoid confusion, when using hexadecimal values you specify the prefix; for example: `allow foo 0x4000,4096`.

allow

`allow` *allow_name start*`,` *len* `[,` *permissions*`]`

Add to the set of addresses DMA may be used to or from. Use the arguments as follows:

- *allow_name* – the name of the allowed DMA address set. This name is a convenience, which you can specifiy to facilitate referring to a memory region (see “*Startup mappings*”).
- *start* – the start address of the allowed range
- *len* – the length, in bytes, of the allowed range
- *permissions* — if specified, may be:
  - `s` – permit the device to use the allowed range as a source of DMA (read permission)
  - `t` – permit the device to use the allowed range as a target of DMA (write permission)

  If neither `s` nor `t` is specified, the `smmuman` service assumes that both are present: permit the device to use the allowed address range as both a source and a target of DMA.

> The recommended method for allocating memory for devices is to use the SMMUMAN API in **libsmmu.a** (see “*SMMUMAN Client API Reference*”. The `smmuman` has the `allow` so it can support legacy drivers (see “*Startup mappings*” in this chapter).

debug

`debug 1|2`

Output debugging information. Specify `2` for more verbose output.

During intialization, all `smmuman` output goes to both `stderr` and `slog2`. Afterwards, `smmuman` output goes to `slog2` only.

device

`device` *device_spec* `[`*allow_name*`]`

Set the current device being configured to *device_spec*.

Note that:

- If no type is specified for *device_spec*, then the default (`mem:`) is assumed. Thus, `device 0xF800B000/4000` is equivalent to `device mem:0xF800B000/4000`.
- If *allow_name* is specified for this option, then the device is restricted to address ranges specified by the `allow` options with the same value for *allow_name* (see `allow` above).

The *device_spec* argument may take the following values:

pci:*

Specify increasingly detailed information about a PCI device, as follows:

- `pci:*` — any PCI device
- `pci:`*bus*:* — any PCI device on bus *bus*
- `pci:`*bus*:*dev*.* — any function number on PCI device *dev* on bus *bus*
- `pci:`*bus*:*dev*.*func* — function *func* on PCI device *dev* on bus *bus*

[mem:]*paddr*[/*length*]

Specify a memory-mapped I/O device at physical address *paddr* for *length* bytes.

Devices may be attached to only one SMMU object at a time. The process that creates the SMMU object and attaches devices to it has exclusive control of the SMMU object and of the memory mappings and memory access permissions for the attached devices.

If you use the startup configuration to map a memory region for a device, the `smmuman` service controls both the SMMU object, and the device's memory mappings and memory access permissions. Your `smmuman` clients won't be able to modify these mappings and permissions, or even remove the device from the SMMU object controlled by the `smmuman` service.

Thus, if you may need to modify a device's memory mappings or memory access permissions after startup, you can specify the device in the startup configuration, but you can't use the `device` *allow_name* argument and assign memory regions to the device.

Memory sharing

The following shows two devices configured to share the same memory region:

```
smmu vtd require

allow foo   0xF8000000,0x4000,st
allow moo   0xF8000000,0x4000,st

device pci:0:18.0 foo
device pci:0:21.0 moo
```

Both the `foo` and `moo` `allow` options specify the same memory region (starting at `0xF8000000`, for `0x4000` bytes), and both have their permissions set to `st`, so devices configured with their *allow_name* arguments set to `foo` or `moo` will have permission to read and write this region.

The `smmuman` configuration syntax provides a better way to achieve the same result, however. You can specify the the same *allow_name* argument for multiple devices (e.g., `foo`), so that all devices with this *allow_name* argument can read and write the same memory region:

```
allow foo   0xF8000000,0x4000,st

device pci:0:18.0 foo
device pci:0:21.0 foo
```

Since this method requires only a single `allow` option entry, it uses less system page memory than the previous configuration.

Finally, if several devices require access to the same memory region but with different permissions, you must use the `allow` option to specify different memory region names. For example, in the configuration below, devices `pci:0:18.0` and `pci:0:21:0` may read from the memory region at `0xF8000000,0x4000`, but only device `pci:0:26.2` may both read and write the region:

```
allow foo   0xF8000000,0x4000,s
allow moo   0xF8000000,0x4000,st

device pci:0:18.0 foo
device pci:0:21.0 foo
device pci:0:26.2 moo
```

foreground

```
foreground
```

After parsing the configuration, leave the `smmuman` service running in the foreground. The default is to leave `smmuman` running in the background.

reserved

`reserved` *start_paddr*`,` *length*

If this option is used, it applies to the device specified by the preceding `device` option.

Some devices may need to access a specific memory region in order to function (e.g., they may need access to code or to data tables found at specific locations on the hardware). For example, a graphics device may store its font definitions separately from the text it will display. This is typical for x86 boards, and for these boards the ACPI tables should provide all the information needed to reserve the required memory regions for whatever devices need them.

Remember: *reserved* memory isn't the same as a *mapped* memory region:

- Reserved memory is only required for some devices, and is used for ancillary data, such as font definitions.
- Typically, the `smmuman` retrieves the required location and size of reserved memory from the board, and programs this information into the IOMMU/SMMUs without your having to specifically ask it to do so.

- Your `smmuman` client should know what memory regions it has asked the `smmuman` service to map for which devices, but the client needs to use the *smmu_device_report_reserved()* function to learn what reserved memory (if any) has been assigned to a DMA device.

You need to use the `reserved` option to manually set aside such regions only if the information isn't available from the board firmware. For example, if on an x86 board the ACPI DMAR tables weren't available, you would need to use the `ignore` option, then specify the VT-d units and the devices. Your configuration might include something like the following:

```
smmu vtd ignore
 unit vtd1 0xfed64000
 unit vtd2 0xfed65000

device pci:0:2.0
 use vtd1
 reserved 0x7b800000,0x4800000

device pci:*
 use vtd2

device pci:0:21.0
 reserved 0x7afe0000,0x20000

device pci:0:21.1
 reserved 0x7afe0000,0x20000
```

Note that no space is permitted between the *start_paddr* and *length* arguments; thus `reserved 0x7afe0000,0x20000` is permitted, but `reserved 0x7afe0000, 0x20000` will fail.

CAUTION:

If a board doesn't provide information it should provide (e.g., the ACPI DMAR tables), the board likely has other problems as well, and you should consult your board manufacturer.

Additionally, any change to your board (e.g., firmware revision) will oblige you to revisit a configuration that uses the `ignore` option:

- x86 – the addresses for the VT-d units may change with any firmware revision; the firmware specification doesn't provide any recommendations or guidelines.
- ARM – the reserved regions for the GPU and for USB support may change if the board RAM layout changes.

For more information about configuring the `smmuman` service for x86 platforms, see "*Options for x86 IOMMUs (VT-ds)*" in this chapter.

safety

```
safety none|warn|required
```

Specify how the `smmuman` service responds if any of the components on which it depends (e.g., `procnto`) isn't a safety-certified component:

none

Ignore the presence of a non-safety component and just run.

warn

If any component isn't safety-certified, issue a warning, and run.

required

If a component isn't safety-certified, issue an error message and move to the DSS (see "*Design Safe State (DSS)*").

Default behaviors are as follows:

- `smmuman` — none
- `smmuman-safety` — required

You can use multiple instances of the `safety` option to specify different responses for different components. For example, in a safety-related system you might use an uncertified support file and have the `smmuman` service issue only a warning for that file, but move to its DSS if any other component isn't certified.

If more than one instance of the `safety` option is specified, the final instance is used to specify the `smmuman` service's global response to the presence of uncertified components (i.e., how it should respond to the presence of an uncertified `procnto` and other components not otherwise explicitly specified in the configuration).

DANGER: Including non-safety SMMUMAN components in a system invalidates the system's safety-certification.

---

set

`set` *var* *val*

The `set` option supports the following variables types:

- `address` – use any of the forms permitted for memory sizes when you specify *val* (e.g., 4K, 0d16384 (see "*About notation*"))
- `boolean` – any of the following values are equivalent; the first term turns on the feature, the second turns it off: `1`/`0`, `yes`/`no`, `on`/`off`
- `number` – any integer
- `string` – a text string

You can use the command line at startup to display currently permitted variables and their contexts. For example:

```
smmuman set ?
 grow-heap   (address, global)
 message-block-timeout  (number, global)
 slogger2-required  (boolean, global)
```

A question mark (`?`) is a shell wildcard character, so you may need to escape it.

grow-heap

- Context: global – applies to the entire service
- Variable type: address
- Default: don't grow the heap

Grow the heap by the amount specified by *val*. For example:

```
set grow-heap 0x4000
```

will increase the heap by 16384 bytes.

message-block-timeout

- Context: global – applies to the entire service
- Variable type: number
- Default: 100 milliseconds

Set the maximum allowed time, in milliseconds, that a message from the `smmuman` service may be blocked before the service sends an unblock pulse to the receiving server. For example:

```
set message-block-timeout 200
```

configures the `smmuman` service to send an unblock pulse to any server that doesn't respond to a message within 200 milliseconds.

The `set` *message-block-timeout* variable must be a value from 5 through 10000, or 0 (zero). A 0 makes the timeout infinite (never time out).

Depending on the server's response (or non-response), the `smmuman` service may terminate with an error. You can use `server-monitor` to handle situations where a server doesn't respond to an unblock pulse (see `server-monitor` in the *QNX Neutrino Utilities Reference*).

slogger2-required

- Context: global – applies to the entire service
- Variable type: boolean
- Default: `on`

Log messages to a `slogger2` buffer.

Currently, all `set` variables for the `smmuman` service are global (i.e., specify the variable once and it applies to the entire service).

smmu

smmu *smmu_type* [*smmu_type_parm*]

Load the **smmu-*smmu_type*.so** support library, and call its initialization routine, passing *smmu_type_parm* to the routine. For example:

```
smmu vtd
```

will load the **smmu-vtd.so** support library needed for x86 boards.

```
smmu rcar3 0xe67b0000,228,0x4f.0x01,0x4f.0x10,0x4f.0x20
```

will load the **smmu-rcar3.so** library with the specified arguments.

For more information about *smmu_type_parm*, see the architecture-specific and board-specific sections in this chapter.

unit

unit *unit_name* [*smmu_unit_parm*]

Define an individual IOMMU/SMMU unit with name *unit_name*. The type of the unit will be from the preceeding `smmu` option.

The unit initialization routine will be invoked and passed *smmu_unit_parm*. For example, for an ARM Renesas R-Car H3 board:

```
smmu rcar3 0xe67b0000,228,0x4f.0x01,0x4f.0x10,0x4f.0x20
 unit vi0 0xfebd0000,14 # video IO domain AXI
 unit vi1 0xfebe0000,15 # video IO domain AXI
 ...
 unit ir  0xff8b0000,3 # IMP domain AXI
 unit hc  0xe6570000,2 # high communication domain AXI
 ...
```

Or, for an x86 board:

```
smmu vtd ignore
 unit vtd1 0xfed64000
 unit vtd2 0xfed65000
```

For more information about *smmu_unit_parm*, see the architecture-specific and board-specific sections in this chapter.

use

use *unit_name* [*smmu_use_parm*]

The current device (specified by the preceeding `device` option) will use the IOMMU/SMMU unit identified by *unit_name*. The *smmu_use_parm* provides additional information on the device to the IOMMU/SMMU unit.

For more information about *smmu_use_parm*, see the architecture-specific and board-specific sections in this chapter.

user

`user` *uid* [**:** *gid* [**,** *sup_gid*] *] | *user_name* [**,** *sup_gid*] *

Set the user ID (*uid*) and group ID (*gid*) and, optionally, supplementary group IDs (*sup_gid*) the `smmuman` service runs with, so that it doesn't have to run as **root**. In the second form, the primary group is the one specified for *user_name* in **/etc/passwd**.

For more information about user and group IDs, see “Process privileges in the *QNX Neutrino Programmer's Guide*.”

Options for ARM SMMUs

On ARM boards, IOMMUs are known as SMMUs; a `smmuman` service running on the hardware needs a support library for the board SMMUs.

The configurations described here are for use with the **smmu-armsmmu.so** and **smmu-armsmmu-safety.so** support libraries for ARM SMMUs. These support libraries implement the code to communicate with ARM SMMUs, as specified in Chapter 16 of *System Memory Management Unit Architecture Specification: SMMU architecture version 2.0 (2016) ARM IHI 0062D.c (ID070116)*. ARM Ltd, 2012-16.

To load one of these support libraries you must set the `smmu` option's *smmu_type* argument to the name of the support library:

- **smmu-armsmmu.so** for non-safety systems
- **smmu-armsmmu-safety.so** for safety-related systems

If you are using Renesas R-Car boards with IPMMUs, see “*Options for Renesas R-Car IPMMUs*” below.

Options

The following describes the options for a `smmuman` service using a **smmu-armsmmu.so** or **smmu-armsmmu-safety.so** support library.

smmu_type_parm

On boards that use the ARM SMMU architecture, the *smmu_type_parm* argument requires no further arguments. For example:

```
smmu armsmmu
```

where `smmu-armsmmu[-safety].so` is the name of the architecture-specific support library.

smmu_unit_parm

On boards that use the ARM SMMU architecture, the syntax for *smmu_unit_parm* is as follows:

paddr **,** *NSclrpt* **,** *NSglrpt* **,** *bypass* [**,** *res_sid* [**,** *cfg_dll*]]

paddr

The physical address of the main memory SMMU registers.

NSclrpt

The interrupt number to use for context faults.

NSglrpt

The interrupt number to use for global faults.

bypass

Bypass unidentified streams. Set to `1` (one) to bypass unidentified streams; set to `0` (zero) to *not* bypass unidentifed streams (see "*Board-specific configuration libraries*" below).

res_sid

Optional. A list of StreamIDs reserved for the system. Entries in the list must be separated by a colon (e.g., `4:5`).

cfg_dll

Optional. The base name for the board-specific configuration DLL (see "*Board-specific configuration libraries*" below).

For example, the following shows how the *smmu_unit_parm* argument might be used for an i.MX8 board:

```
unit mmu500 0x51400000,64,66,1,4:5:16,imx8
```

where `mmu500` is the name assigned to the SMMU unit in the configuration (see "*Global options*"), `0x514000000` is the physical address of the SMMU unit registers, `64` is the number of the interrupt to assert when a context fault occurs, `66` is the number of the interrupt to assert when a global fault occurs, `1` instructs `smmuman` to bypass unidentified streams, `4:5:16` specifies that StreamIDs 4, 5 and 16 are reserved for the system, and `imx8` instructs `smmuman` to load the **smmu-cfg-imx8.so** library.

smmu_use_param

On boards with ARM SMMU architectures, the syntax for *smmu_use_parm* is as follows:

`smmu_use_param` *sid*

sid

The ID of the stream (StreamID in ARM nomenclature) the device uses to perform the transaction.

Set to `*` to set the StreamID at runtime. If you set StreamIDs at runtime, you must use *smmu_unit_parm res_sid* to identify StreamIDs that are reserved and therefore can't be used when setting a StreamID at runtime.

For example, the following shows how the *smmu_use_parm* argument might be used for an i.MX8 board, with the StreamID set in the configuration:

```
device mem:0x5b040000 use mmu500 9
```

where `mmu500` is the name assigned to the SMMU unit in the configuration, and `9` is the StreamID.

The following shows how the *smmu_unit_parm* and *smmu_use_param* arguments might be uses for an i.MX8 board, with the StreamID set at runtime:

StreamID set at runtime

The following shows part of the configuration for an i.MX8 board, with the StreamID for a device set at runtime:

```
## StreamID is 0 is reserved for bypass.
smmu smmu-armsmmu.so
 unit mmu500 0x51400000,64,66,1,0,imx8
...
device mem:0x2c000000 use mmu500 *
```

where `smmu smmu-armsmmu.so` identifies the architecture-specific support library: `unit mmu500 0x51400000,64,66,1,0,imx8` configures the SMMU unit, specifying that StreamID 0 is reserved, and `device mem:0x2c000000 use mmu500 *` specifies that the StreamID for this device should be set at runtime.

Board-specific configuration libraries

Some boards that use ARM SMMUs require board-specific SMMU settings. If the *cfg_dll* argument is specified in *smmu_unit_param*, the **smmu-armsmmu.so** support library will load the specifed configuration library, which looks after the board-specific settings.

These libraries with board-specific settings are named as follows: **smmu-cfg-*board*.so**, where *board* is the board name.

For example, on NXP i.MX8 platforms, StreamIDs are configurable. Use the *smmu_unit_param cfg_dll* argument to have **smmu-armsmmu.so** load the **smmu-cfg-imx8.so** library, which will set the StreamIDs according to the configuration you specify in the `smmuman` configuration file.

Options for Renesas R-Car IPMMUs

On the Renesas R-Car H3 and other related boards, IOMMU/SMMUs are known as IPMMUs; a `smmuman` service running on the hardware needs a support library for the board IPMMUs.

The configurations described here are for use with the **smmu-rcar3.so** and **smmu-rcar3-safety.so** support libraries for Renesas R-Car H3 IPMMUs. These support libraries implement the code to communicate with Renesas R-Car H3 IPMMUs, as specified in Chapter 16 *Renesas R-Car Series, 3rd Generation User's Manual: Hardware*, Nov. 2018 (Rev. 1.50).

To load this support library, you must set the `smmu` option's *smmu_type* argument to the name of the support library:

- **smmu-rcar3.so** for non-safety systems
- **smmu-rcar3-safety.so** for safety-related systems

Options

The following describes the options for a `smmuman` service using a **smmu-rcar3.so** or **smmu-rcar3-safety.so** support library for SMMUs that use the Renesas R-Car H3 IPMMU architecture.

smmu_type_parm

On boards that use the Renesas R-Car H3 IPMMU architecture, the *smmu_type_parm* argument is as follows:

paddr **,** *fault_vector* **,** *mm_fault_bit* { **,** *socid* **.** *revid* }

paddr

The physical address of the main memory IPMMU unit registers.

fault_vector

The interrupt that occurs when an illegal DMA request is attempted.

mm_fault_bit

The interrupt bit that is set when a page table fault occurs.

The page table fault interrupt bit may be different on different chips and chip revisions; check the hardware documentation for the chip you are using.

socid

A SoC identification number supported by the configuration data.

revid

A supported revision number of the SoC supported by the configuration data.

For example, the following configuration is for an R-Car H3 SOC (revision 3.0), as described in the *Renesas R-Car Series, 3rd Generation User's Manual: Hardware*, Nov. 2018 (Rev. 1.50):

```
smmu rcar3 0xe67b0000,228,18,0x4f.0x20
```

where `0xe67b0000` is the physical address of the main memory IPMMU unit registers, `228` is the interrupt that occurs when an illegal DMA request is attempted, `18` is the page table fault interrupt bit, and `0x4f.0x20` is the ID and revision numbers of the SoC supported by the configuration data.

smmu_unit_parm

On boards that use the Renesas R-Car H3 IPMMU architecture, the syntax for *smmu_unit_parm* is as follows:

paddr **,** *intr_status_bit*

paddr

The physical address of the IPMMU unit registers.

intr_status_bit

The bit number in the interrupt status register where this unit has encountered a fault condition.

smmu_use_parm

On boards that use the Renesas R-Car H3 IPMMU architecture, the syntax for *smmu_use_parm* is as follows:

utlb

utlb

The micro-translation lookaside buffer (TLB) number to use for the device.

For example:

```
device mem:0xE6EF0000 use vi0 0
```

where `mem:` defines a memory-mapped I/O device at physical address `0xE6EF0000`; this device uses the `vi0` IOMMU/SMMU unit and the micro-translation lookaside buffer (TLB) `0`.

Options for x86 IOMMUs (VT-ds)

On x86 boards, IOMMU/SMMUs are known as VT-ds; configuration is needed only for non-standard VT-d variants.

The configurations described here are for use with the **smmu-vtd.so** and **smmu-vtd-safety.so** support libraries for IOMMUs with the x86 VT-d architecture. These support libraries implement the code to communicate with x86 VT-ds, as specified in *Intel Virtualization Technology for Directed I/O Architecture Specification.* Intel, Nov. 2017 (D51397-009, Rev. 2.5).

To load these support libraries, you must set the `smmu` option's *smmu_type* argument to the name of the support library:

- **smmu-vtd.so** for non-safety systems
- **smmu-vtd-safety.so** for safety-related systems

Options

The following describes the options for a `smmuman` service using a **smmu-vtd.so** or **smmu-vtd-safety.so** support library for IOMMUs that use the x86 VT-d architecture.

smmu_type_parm

On boards with Intel VT-d IOMMU hardware, the syntax for *smmu_type_parm* is as follows:

```
require | ignore
```

require

The VT-d information must be present in the ACPI tables.

ignore

Ignore ACPI information.

CAUTION: Use `ignore` only if the board firmware is incorrect for your board design and you need to manually input the VT-d unit and device information that would normally be found in these tables. Consult your board manufacturer's documentation. For example, see “Intel Virtualization Technology for Directed I/O” available from *software.intel.com/en-us/articles/intel-sdm*.

If neither `required` nor `ignore` is specified, the `smmuman` service doesn't require the ACPI tables, but uses them if they are present.

smmu_unit_parm

On boards with Intel VT-d IOMMU hardware, the syntax for *smmu_type_parm* is as follows:

vtd_paddr

vtd_paddr

The base physical address of the VT-d device registers for the unit.

smmu_use_parm

On boards with Intel VT-d IOMMU hardware, no value is required or permitted for *smmu_use_parm*.

The most common way to start `smmuman` for x86 systems is:

```
smmuman smmu vtd
```

This startup configures `smmuman` to use the default ACPI tables, and the default locations for VT-d.

Mapping DMA devices and memory regions

The preferred method for mapping DMA devices to memory regions and specifying their access permissions is through the SMMUMAN client API.

After the `smmuman` service has started, processes such as device drivers or external applications can use the client API to connect to the service as clients and use it to program the system IOMMU/SMMUs. This API is defined in the **smmu.h** header file and made available in the **libsmmu.a** library. Clients can use this API to:

- add or remove DMA devices
- add or remove memory regions that DMA devices are permitted to access, or modify the devices' access permissions
- have the `smmuman` service inform them of illegal DMA device attempts to access memory

CAUTION: Until the `smmuman` service programs the IOMMU/SMMUs on a board with a DMA device's memory access permissions, that device has unrestricted access to memory.

Task overview

A `smmuman` client using the service to map DMA devices and memory regions typically does the following:

1. Obtains the custom abilities it will need to connect to the `smmuman` (see "*Connecting to the service*" below).
2. Calls *smmu_init()* to *connect to the `smmuman` service*.
3. Calls *smmu_obj_create()* to *create SMMU objects*.
4. Calls the *smmu_device_add_*()* functions to *add* DMA devices to the SMMU objects.
5. Calls *smmu_mapping_add()* to *add the memory mappings and permissions* appropriate for the DMA devices attached to the SMMU objects.
6. As needed, calls *smmu_mapping_add()* or the appropriate *smmu_device_add_*()* function to *remove devices or memory mappings* that are no longer needed.
7. Calls *smmu_obj_destroy()* to destroy SMMU objects that are no longer needed and, finally, calls *smmu_fini()* to *terminate the connection* with the `smmuman` service.

See *Monitoring DMA transgressions* below for instructions on how to have the `smmuman` service notify a client of illegal DMA device attempts to access memory.

Connecting to the service

To use the SMMUMAN API, a process in your system must become a client of the `smmuman` service.

Before initializing contact with the service, the process must obtain the appropriate custom abilities:

- SMMU_ABILITY_ATTACH_NAME (`"smmu/attach"`) – required for all connections to the service
- SMMU_ABILITY_TARGET_NAME (`"smmu/target"`) – required if the client will use the SMF_TARGET flag when calling *smmu_mapping_add()* (currently reserved for the QNX Hypervisor)

Use *procmgr_ability_lookup()* and *procmgr_ability()* to look up and control these custom abilities. For more information, see *smmu_init()* in the *SMMUMAN Client API Reference* chapter, and *procmgr_ability_lookup()* and *procmgr_ability()* in the *C Library Reference*.

After your process has the required abilities, call *smmu_init()* to check if the `smmuman` service is running and connect to it as a client.

Checking the service's safety status

If you are running a safety-related system, you must use the `smmuman` safety variant. A `smmuman` client can call *smmu_safety()* to check if the `smmuman` service to which it is connected is the safety variant.

See *safety* in "*Global options*" for information about configuring your system's response to the presence of components that aren't safety-certified.

Creating SMMU objects

The `smmuman` service manages dynamic memory region mappings, and DMA device mappings and permissions through SMMU objects. Thus, at any time after it has initiated contact with the `smmuman` service and become its client, your process can call *smmu_obj_create()* to create a SMMU object to which it can then add memory mappings and DMA devices.

A `smmuman` client may create as many SMMU objects as it needs; this allows different devices to have different mappings to memory regions. For example, the QNX Hypervisor `qvm` processes use this capability.

Additionally, at any time while the `smmuman` service is running, your client can remove a DMA device from a SMMU object by calling the relevant *smmu_device_add_*()* function with the *sop* argument set to NULL.

Adding DMA devices

After your `smmuman` client has created one or more SMMU objects, it can add devices to the objects. Call one of the following for each DMA device that needs to access memory:

- *smmu_device_add_generic()* — add a device of an unspecified type to a SMMU object
- *smmu_device_add_mmio()* — add a Memory-Mapped I/O (MMIO) device to a SMMU object
- *smmu_device_add_pci()* — add a Peripheral Component Interconnect (PCI) device to a SMMU object

If you are adding a device to an object after you have added a memory mapping to that object, you may need to refresh the memory mappings (see "*Preferred sequence for adding memory mappings and devices*").

All devices attached to a SMMU object are granted access to the memory regions mapped to that object, with the same permissisons. If you want to give two devices access to the same memory region, but with different permissions (e.g., one device may only read, the other may only write), then you must create two separate SMMU objects (see *smmu_mapping_add()*).

A device may have only one owner. Attempting to use one of the *smmu_device_add_*()* functions when the device has already been added to a SMMU object of a different client will result in an EBUSY error. A `smmuman` client may move a device from one of its SMMU objects to another one of its SMMU objects, however, because this action doesn't change the device owner.

Adding memory mappings

After your `smmuman` client has created one or more SMMU objects, it can add memory regions to these objects, specifying the access permissions that will be applied to the devices attached to each object. Call *smmu_mapping_add()* to add a memory region to an object, so that devices attached to the object can have access to this memory region.

Preferred sequence for adding memory mappings and devices

The preferred sequence for adding memory regions and devices to a SMMU object is to add the devices first, then add the memory regions. These memory regions will apply to all the devices linked to the SMMU object.

This is not always possible, however, and you may need to add the memory region mappings first. For example, if a USB device is added to your system after startup, you may need to add it to a memory region mapping you created earlier.

If you add the mappings first, and then you add a device, the *smmu_device_add_*()* function may return `-1` (failure), `0` (nothing more to do), or `+1` (see *smmu_device_add_generic()* and the other *smmu_device_add_*()* functions).

A `+1` return from these functions means that the hardware module used for the new device has indicated that page tables for this device are needed. You address this by calling *smmu_mapping_add()* again for all the active mappings on the SMMU object.

After a `+1` return from a *smmu_device_add_*()* function:

- Devices that were previously added to the SMMU object may continue their DMA operations; they don't have to wait on the call to *smmu_mapping_add()*.
- You need to add only the active mappings again; you don't need to do anything about mappings you deleted from the SMMU object.

Removing devices and memory regions

If a DMA device is no longer on the system (e.g., USB device has been removed), you can call the appropriate *smmu_device_add_*()* function with its *sop* argument set to NULL to *Adding DMA devices* remove the device from a SMMU object.

If the devices attached to a SMMU object no longer require access to some memory regions mapped to a SMMU object, you can remove these regions from the object by calling *smmu_mapping_add()* without specifying any of the SMF_READ, SMF_WRITE, or SMF_EXEC permissions for the *flags* argument (i.e., the *flags* argument set to SMF_NONE; see *`smmu_mapping_flags`* in the "*SMMUMAN Client API Reference*").

Disconnecting your `smmuman` client

If your client no longer needs to interface with the `smmuman` service, it can disconnect from it. To disconnect from the `smmuman` service, a client must:

1. For every SMMU object it has created, call *smmu_obj_destroy()* to destroy each SMMU object in turn.
2. After it has destroyed all its SMMU objects, call *smmu_fini()* to terminate its connection with the `smmuman` service.

Monitoring DMA transgressions

After the `smmuman` service programs the memory mappings for DMA devices into the system IOMMU/SMMUs, these units will inform it of any attempts by these devices to access memory outside their assigned regions.

A `smmuman` client can register to be informed of these transgressions. This client doesn't have to be the client that owns the SMMU objects and has added devices and mapped memory regions to it. You could have some clients (e.g., device drivers) create objects, add devices and map memory regions, and another client (i.e., some other process in your system) monitor transgressions.

1. Call *smmu_xfer_notify()* to register for updates.
2. Call *smmu_xfer_status()* at any time to query the `smmuman` service for any records it may have of attempted transgressions.
3. Call *smmu_xfer_notify()* to reregister for updates.

When the `smmuman` service informs a client that an IOMMU/SMMU has refused a DMA device's memory access attempt, it clears that client's registration for notification of DMA device memory access transgressions. The *smmu_xfer_notify()* call after the *smmu_xfer_status()* call is required to reregister the client so it will receive notifications of any subsequent attempted transgressions.

Startup mappings

The smmuman configuration can be used to create DMA device to memory region mappings and permissions that the smmuman service reads in when it starts up.

Startup mappings, or client-side API?

When you use the smmuman client-side API to manage DMA device memory mappings and permissions, the smmuman clients (e.g., drivers and other processes) communicate directly with the smmuman service. When you use the smmuman configuration to specify DMA device memory mappings and permissions, you are setting up *indirect* communication between the DMA device drivers and the smmuman service. Neither the drivers nor the service know about each other. You need to provide the information they require to work together indirectly, through the smmuman configuration, and the startup for each process that will become a smmuman client (e.g., DMA device driver startup).

Depending on the needs of your system, you may want or need to use mappings configured at startup for all the DMA devices and their mapped memory regions on your system, and never use the client-side API.

You may also include in the startup mappings only devices needed immediately at startup, leaving the rest to be added later through the smmuman client-side API, as needed, by DMA device drivers and other processes.

If you are using a legacy DMA device driver (i.e., a driver that pre-dates smmuman and hence doesn't use the smmuman client-side API), you must use the smmuman configuration to create DMA device to memory region mappings and permissions when the smmuman service starts.

Typed memory support

To create the mappings in your smmuman configuration you use typed memory, because typed memory regions can be named. The smmuman configuration uses these names to map memory regions to devices. Thus, your DMA device driver must have an option that allows you to specify typed memory. These options differ from driver to driver (see "*Starting the drivers*" for some examples).

If the driver doesn't have such an option, you need to modify the driver. If this is the case, then unless you have a compelling reason not to do so, you should modify the driver to use the smmuman client-side API and use this API to manage your DMA device memory mappings and permissions (see "*Mapping DMA devices and memory regions*" in this chapter).

For more information about typed memory, see "Typed Memory" in the QNX Neutrino OS *System Architecture* guide.

Tasks

Using the smmuman configuration to create DMA device to memory region mappings and permissions involves the following tasks:

1. *Configuring the smmuman service*
2. *Setting aside the typed memory regions at system startup*
3. *Starting the drivers*

When you specify devices, memory regions and their permissions in the `smmuman` configuration, you are simply instructing the `smmuman` service to act as its own client and use its client-side API to map these devices and memory regions.

Configuring the smmuman service

To use the `smmuman` configuration to create mappings between DMA devices and memory regions, add to the `smmuman` configuration:

- the typed memory regions needed for each device, specifying each region's name (*allow_name*)
- the devices, specifying by name (*allow_name*) the memory region to which the device should be mapped

For example:

```
debug 2

smmu vtd require

allow smbs0   $asinfo_start{smbs0},$asinfo_length{smbs0},st
allow xhci    $asinfo_start{xhci},$asinfo_length{xhci},st
allow hsu0    $asinfo_start{hsu0},$asinfo_length{hsu0},st
allow hsu1    $asinfo_start{hsu1},$asinfo_length{hsu1},st
allow hsu2    $asinfo_start{hsu2},$asinfo_length{hsu2},st
allow ieheci0 $asinfo_start{ieheci0},$asinfo_length{ieheci0},st
allow sdhci   $asinfo_start{sdhci},$asinfo_length{sdhci},st
allow ixgbe   $asinfo_start{ixgbe},$asinfo_length{ixgbe},st

device pci:0:18.0 smbs0
device pci:0:21.0 xhci
device pci:0:26.0 hsu0
device pci:0:26.1 hsu1
device pci:0:26.2 hsu2
device pci:0:27.0 ieheci0
device pci:0:28.0 sdhci
device pci:2:00.1 ixgbe
device pci:2:00.0 ixgbe
```

Note that the `allow` option's first argument is *allow_name*, and that the `device` uses this name to associate the device with a memory region. In the example above, one mapping (`hsu0`) is in bold.

The example uses textual substitutions and places the configuration values in the system page tables (see "*Textual substitutions*" in this chapter). For more information about how to manage permissions, see the *`allow`* and *`device`* options under "*Global options*" in this chapter.

Setting aside the typed memory regions at system startup

In your system build file, configure your startup program to use the `-R` option to set aside the typed memory regions you will use for your DMA devices. For example, below is the startup for an x86 board that uses the `smmuman` service configured as in "*Configuring the smmuman service*" above:

```
### Startup suitable for use with SMMUMAN (with smbus, hsu, IE heci and ethernet)
startup-foo -vv -T  \
-R64K,4K,smbs0 -R8K,4K,hsu0 -R8K,4K,hsu1 -R8K,4K,hsu2 -R128K,4K,ieheci0 -R16M,4K,ixgbe\
-f1700M,1700000000
```

Note that each instance of the `-R` option specifies the size of the memory region, its alignment, and a name. This name is the name used in the `smmuman` configuration to identify each memory region. In the startup configuration above, the memory region identified in bold is the region similarly identified in the `smmuman` configuration.

Startup programs are included in the QNX BSPs. For more information about startup programs, see "Startup Programs" in *Building Embedded Systems*. For more information about the `-R` startup option, see "`startup-*` options" in the *Utilities Reference*.

Starting the drivers

Finally, after you have specified the DMA device to typed memory mappings in the `smmuman` configuration, and reserved the typed memory regions in the startup program, you need to start each device driver, specifying by name the typed memory region it has been assigned.

Below are some examples. Notice that the options for specifying the named memory region differ from device to device. In each example the option and the typed memory region name are shown in bold; the name corresponds to a region specified in the startup program and by the `smmuman` configuration `allow` option's *allow_name* argument.

System manager bus driver

```
io-smb -v -t smbs0 -M slave -i addr=0x46,fifo=5 -c 0x45 -a 0x44
```

Serial driver

```
devc-serhsu vid=0x8086,did=0x19d8,pci=0,dma=hsu0 -u 1 -e -b115200
```

Networking stack and ixgbe driver

```
io-pkt-v6-hc -ptcpip pkt_typed_mem=ixgbe
mount -T io-pkt -o pci=1,ign_cksum,mac=001122330001, \
typed_mem=ixgbe devnp-ixgbe.so
```

eMMC driver

```
devb-sdmmc blk cache=1m cam bounce=128K mem name=sdhci
```


USB and USB mass storage drivers

```
io-usb-otg -t memory=xhci -d xhci pindex=0,memory=xhci &
waitfor /dev/usb/io-usb-otg
devb-umass cam pnp bounce=128k mem name=xhci blk memory=xhci &
```

SMMUMAN in a QNX Hypervisor system

The smmuman service is required to support pass-through DMA devices in a QNX Hypervisor system.

In a hypervisor system, the smmuman service is required, not just to program the DMA device memory regions and permissions into the board IOMMU/SMMUs, but also to manage guest-physical memory to host-physical memory translations and access for DMA devices passed through to a guest running in a hypervisor VM (see "*SMMUMAN in a QNX Hypervisor guest*" in the "*Architecture*" chapter).

These memory translations require hardware support for virtualization, specifically, two-stage IOMMU/SMMUs that can map the two translation levels.

Each VM (qvm process instance) in the hypervisor host configures the smmuman service to program the board IOMMU/SMMUs to limit the host-physical memory to which DMA devices passed through to its guest have access. Any DMA device passed-through to a guest is limited to the host-physical memory region(s) assigned to that VM in its configuration (see the *QNX Hypervisor User's Guide*).

This configuration protects the hypervisor host and guests running in other hypervisor VMs from errant DMA devices in the guest, but it doesn't protect the guest from its own DMA devices (DMA devices passed through to that guest).

The guest must configure its own smmuman service with the memory regions and permissions it requires for each pass-through DMA device it controls. The smmuman service running in the guest uses the smmu virtual device (**vdev-smmu.so**) to pass on these memory regions and permissions to the smmuman service running in the host, which in turn programs the board IOMMU/SMMUs.

The figure below illustrates how the smmuman service is implemented in a QNX Hypervisor guest to constrain a pass-through DMA device to its assigned memory region:

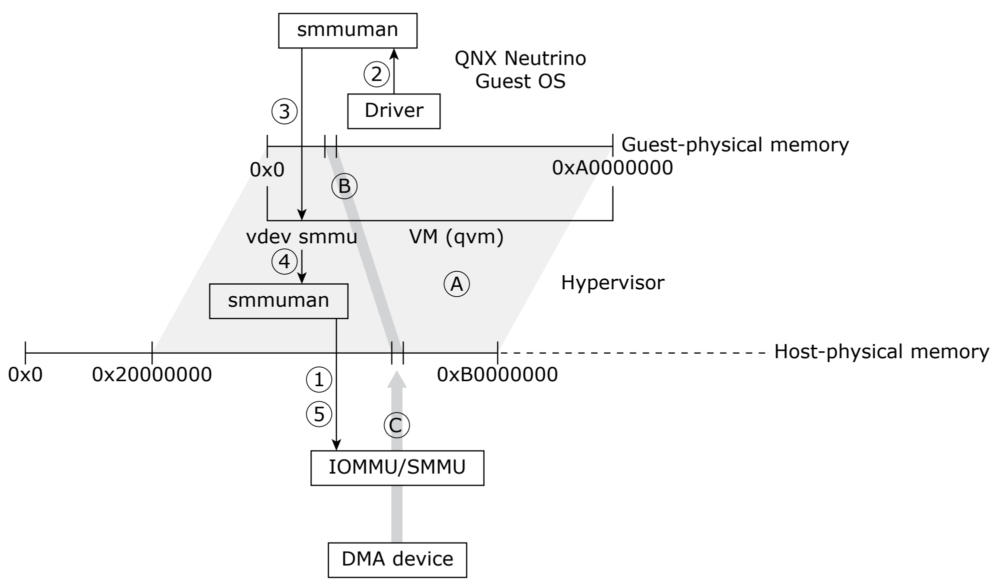

Figure 4: The smmuman service running in a guest uses the smmuman service running in the hypervisor.

For simplicity, the diagram shows a single guest with a single pass-through DMA device:

1. In the hypervisor host, the VM (`qvm` process instance) that will host the guest uses the `smmuman` service to program the IOMMU/SMMU with the memory range and permissions (A) for the entire physical memory regions that it will present to its guest (e.g., `0x20000000` to `0xB0000000`).

   This protects the hypervisor host and other guests if DMA devices passed through to the guest erroneously or maliciously attempt to access their memory. It doesn't prevent DMA devices passed through to the guest from improperly accessing memory assigned to that guest, however.

2. In the guest, the driver for the pass-through DMA device uses the SMMUMAN client-side API to program the memory allocation and permissions (B) it needs into the `smmuman` service running *in the guest*.
3. The guest's `smmuman` service programs the `smmu` virtual device running in its hosting VM as it would an IOMMU/SMMU in hardware: it programs into the `smmu` virtual device the DMA device's memory allocation and permissions.
4. The `smmu` virtual device uses the client-side API for the `smmuman` service running in the hypervisor host to ask it to program the memory allocation and permissions (B) requested by the guest's `smmuman` service into the board's IOMMU/SMMU(s).
5. The host's `smmuman` service programs the pass-through DMA device's memory allocation and permissions (B) into the board IOMMU/SMMU.

The DMA device's access to memory (C) is now limited to the region and permissions requested by the DMA device driver in the guest (B), and the guest OS and other components are protected from this device erroneously or maliciously accessing their memory.

You can't use *startup mappings* in the configuration for the `smmuman` service running in the hypervisor host to map DMA devices that are passed through to a guest in a hypervisor VM. You can use startup mappings in the configuration for the `smmuman` service running in a guest, however.

Chapter 3
SMMUMAN Client API Reference

This chapter describes the API that SMMUMAN clients can use to access and manage the `smmuman` services.

This API is defined in the **smmu.h** public header file.

smmu_* data structures

Data structures used by the SMMUMAN client API

The following data structures are defined in the **smmu.h** header file for the SMMUMAN client API.

- *smmu_devid*
- *smmu_devid_mmio*
- *smmu_devid_pci*
- *smmu_map_entry*
- *smmu_status*

smmu_devid

```
union smmu_devid {
    unsigned  type;
    struct smmu_devid_pci pci;
    struct smmu_devid_mmio mmio;
    _Uint64t  __spacer[4];
};
```

The `smmu_devid` data structure is used to identify devices. Its members include:

type

The type of device; can be one of the values defined by the `smmu_devid_type` enumerated values.

pci

If *type* is set to SDT_PCI, the `smmu_devid_pci` data structure specifying the PCI device information.

mmio

If *type* is set to SDT_MMIO, the `smmu_devid_mmio` data structure specifying the MMIO device information.

__spacer

For internal use.

smmu_devid_mmio

```
struct smmu_devid_mmio {
 unsigned type;
 unsigned length;
 _Uint64t phys;
};
```

The `smmu_devid_mmio` data structure is used to identify MMIO devices. Its members include:

type

The type of device; should be SDT_MMIO

length

The length of the memory region the device requires for its registers.

phys

The physical address of the MMIO device's register memory region.

smmu_devid_pci

```
struct smmu_devid_pci {
 unsigned type;
 unsigned bus;
 unsigned dev;
 unsigned func;
};
```

The `smmu_devid_pci` is used to identify PCI devices. Its members include:

type

The type of device; should be SDT_PCI

bus

The PCI device's bus number.

dev

The PCI device's device number.

func

The PCI device's function number.

If you set *bus*, *dev*, or *func* to SMMU_PCI_FIELD_ANY, any value is allowed.

smmu_map_entry

```
struct smmu_map_entry {
 union {
  void  *virt;
  _Uint32t virt32;
  _Uint64t virt64;
  _Uint64t phys;
 };
 _Uint64t len;
};
```


The `smmu_map_entry` data structure is used by the *smmu_device_report_reserved()* and *smmu_mapping_add()* functions; it includes the following members:

virt

For future use.

virt32

For future use.

virt64

For future use.

phys

The address of the DMA device in physical memory.

len

The length (in bytes) of the DMA device memory region.

For more information about guest-physical and host-physical addresses and memory, see the “QNX Virtual Environments” chapter in the *QNX Hypervisor User's Manual*.

smmu_status

```
struct smmu_status {
 _Uint64t  hw_specific;
 _Uint64t  fault_addr;
 union smmu_devid devid;
 unsigned  flags;
};
```

The `smmu_status` data structure is used by the *smmu_xfer_status()* function; it carries information about DMA device transgressions, and includes the following members:

hw_specific

Hardware-specific information.

fault_addr

The physical address in memory the DMA tried to access, triggering the fault.

devid

The ID of the misbehaving DMA device.

flags

A permissions flag indicating the reason for a failed request (see *smmu_mapping_flags*).

If the SSF_DROPPED_FAULTS bit is set, the `smmuman` service had to drop one or more transgression records before the client retrieved them.

Enumerated values and constants

Enumerated values used by the SMMUMAN client API

The following enumerated values and constants are defined in the **smmu.h** header file for the SMMUMAN client API.

SMMU_ABILITY_* constants

```
#define SMMU_ABILITY_ATTACH_NAME "smmu/attach"
#define SMMU_ABILITY_TARGET_NAME "smmu/target"
```

These abilities must be obtained before calling *smmu_init()*.

Constant	Value	Meaning
SMMU_ABILITY_ATTACH_NAME	"smmu/attach"	Ability required to connect as a client of the smmuman service (see *smmu_init()*).
SMMU_ABILITY_TARGET_NAME	"smmu/target"	Ability required to use the SMF_TARGET flag when calling *smmu_mapping_add()*. Currently reserved for the QNX Hypervisor.

smmu_devid_type

```
enum smmu_devid_type {
 SDT_PCI,
 SDT_MMIO,
 SDT_NUM_TYPES,
};
```

The smmu_devid enumerated values specify the DMA device type. You can use these values when calling a *smmu_device_add_*()* function. The specified types include:

- SDT_PCI — the device is a PCI device
- SDT_MMIO — the device is an MMIO device
- SDT_NUM_TYPES — the number of different types

smmu_mapping_flags

```
enum smmu_mapping_flags {
 SMF_NONE = 0x00,
 SMF_READ = 0x01,
 SMF_WRITE = 0x02,
 SMF_EXEC = 0x04,
 SMF_VIRT = 0x08,
```

```
  SMF_TARGET = 0x10,
};
```

The `smmu_mapping_flags` enumerated values specify bitmaps with the permissions a `smmuman` service client requests the service to assign to a DMA device. These permissions include:

- SMF_NONE (`0x00`) — no permissions
- SMF_READ (`0x01`) — the DMA device may only read from the region
- SMF_WRITE (`0x02`) — the DMA device may write to the region
- SMF_EXEC (`0x04`) — the DMA device may execute from the region
- SMF_VIRT (`0x08`) — reserved for future use
- SMF_TARGET (`0x10`) — respect the value of the *smmu_mapping_add()* function's *target* argument

  Do not set. Currently reserved for the QNX Hypervisor; the client must have the SMMU_ABILITY_TARGET_NAME custom ability.

Renesas R-Car IPMMUs don't allow the specification of write-only regions (i.e., write, but not read). With this hardware, if you assign a memory block write permission, read permission is also implicitly specified.

---

smmu_obj_create_flags

```
enum smmu_obj_create_flags {
  SOCF_NONE    = 0x0000,
  SOCF_RESERVED_MANUAL = 0x0001,
};
```

The `smmu_obj_create_flags` enumerated values specify whether it is the `smmuman` service or the current client that looks after adding and removing any reserved memory regions required by the devices the client adds to a SMMU object:

- SOCF_NONE (`0x0000`) — no flags are specified; the `smmuman` service looks after adding and removing any reserved memory regions required by the devices a client adds to a SMMU object
- SOCF_RESERVED_MANUAL (`0x0001`) — the `smmuman` client is responsible for adding and removing any reserved memory regions required by the devices it adds to a SMMU object

Currently used only by the QNX Hypervisor. For more information about reserved memory regions, see the *`reserved`* option under "*Global options*".

SMMU_PCI_FIELD_ANY

```
#define SMMU_PCI_FIELD_ANY (~0u)
```

The SMMU_PCI_FIELD_ANY constant can be used to allow any value for a PCI bus, device, or function number (see *smmu_device_add_pci()*).

SMMU_SAFETY_* constants

For future use.

SSF_DROPPED_FAULTS

```
#define SSF_DROPPED_FAULTS 0x80000000u
```

SSF_DROPPED_FAULTS specifies the bit that the `smmuman` service sets in the *`smmu_status`*'s `flag` member to indicate that it had to drop one or more transgression records (see *smmu_xfer_status()*).

smmu_device_add_generic()

Add a device to a SMMU object, or remove it

Synopsis:

```
#include <smmu.h>

int smmu_device_add_generic(struct smmu_object *sop,
                             const union smmu_devid *devid);
```

Arguments:

sop

Pointer to the SMMU object to which the device referenced by *devid* will be added, or NULL to remove the device from any SMMU objects owned by the current client.

devid

Pointer to the ID of the device to be added to or removed from a SMMU object.

Library:

libsmmu.a

Description:

The *smmu_device_add_generic()* function is called by the *smmu_device_add_mmio()* and *smmu_device_add_pci()* convenience functions. It can also be used directly by a `smmuman` client.

The *smmu_device_add_generic()* function updates the `smmu_object` data structure referenced by *sop* with a pointer to a device ID. This adds the device to the SMMU object, so that the `smmuman` client that created the SMMU object can program the hardware IOMMU/SMMUs to allow the device to access the memory regions associated with the SMMU object.

All devices attached to a SMMU object are granted access to the memory regions mapped to that object, with the same permissisons. If you want to give two devices access to the same memory region, but with different permissions (e.g., one device may only read, the other may only write), then you must create two separate SMMU objects (see *smmu_mapping_add()*).

A device may have only one owner. Attempting to use one of the *smmu_device_add_*()* functions when the device has already been added to a SMMU object of a different client will result in an EBUSY error. A `smmuman` client may move a device from one of its SMMU objects to another one of its SMMU objects, however, because this action doesn't change the device owner.

To remove a device from SMMU objects, call this function with the *sop* argument set to NULL.

Returns:

-1

Failure: *errno* is set.

EBUSY

Failure when attempting to add to a SMMU object a DMA device that is owned by another `smmuman` client.

ENOENT

Failure: when attempting to:

- add a device – the `smmuman` service doesn't know which IOMMU/SMMU unit controls the device being added
- remove a device – the `smmuman` client making the call doesn't own the SMMU object to which the device is attached

0

Success

+1

Success, but the client must call *smmu_mapping_add()* to reissue the memory mappings for this object (see "*Preferred sequence for adding memory mappings and devices*" for more information about why memory mappings may need to be reissued).

smmu_device_add_mmio()

Add an MMIO device to a SMMU object, or remove it

Synopsis:

```
#include <smmu.h>

int smmu_device_add_mmio(struct smmu_object *const sop,
                         _Uint64t const phys,
                         unsigned const len);
```

Arguments:

sop

Pointer to the SMMU object to which the device will be attached, or NULL to remove the device from any SMMU objects owned by the current client.

phys

The physical address of the device to be attached to or removed from a SMMU object.

len

The number of bytes for device registers.

Library:

libsmmu.a

Description:

The *smmu_device_add_mmio()* function is a convenience function. It calls *smmu_device_add_generic()* to add a Memory-Mapped I/O (MMIO) device to a SMMU object.

To remove an MMIO device from SMMU objects, call this function with the *sop* argument set to NULL.

Returns:

-1

Failure: *errno* is set.

EBUSY

Failure when attempting to add to a SMMU object a DMA device that is owned by another `smmuman` client.

ENOENT

Failure: when attempting to:

- add a device – the `smmuman` service doesn't know which IOMMU/SMMU unit controls the device being added
- remove a device – the `smmuman` client making the call doesn't own the SMMU object to which the device is attached

0

Success

+1

Success, but the client must call *smmu_mapping_add()* to reissue the memory mappings for this object (see "*Preferred sequence for adding memory mappings and devices*" for more information about why memory mappings may need to be reissued).

smmu_device_add_pci()

Add a PCI device to a SMMU object, or remove it

Synopsis:

```
#include <smmu.h>

int smmu_device_add_pci(struct smmu_object *const sop,
                        unsigned const pbus,
                        unsigned const pdev,
                        unsigned const pfunc);
```

Arguments:

sop

Pointer to the SMMU object to which the device will be attached, or NULL to remove the device from any SMMU objects owned by the current client.

pbus

The PCI device's bus number.

pdev

The PCI device's device number.

pfunc

The PCI device's function number.

For *pbus*, *pdev*, and *pfunc*, you can use the *SMMU_PCI_FIELD_ANY* constant for wildcarding.

CAUTION:

If you use wildcarding for all bus, device and function combinations, *smmu_device_add_pci()* checks the buses, devices, and functions on the entire system, which may take a long time.

To avoid this, use wildcarding for only one or two of bus, device, function.

Library:

libsmmu.a

Description:

The *smmu_device_add_pci()* function is a convenience function. It calls *smmu_device_add_generic()* to add a Peripheral Component Interconnect (PCI) device to a SMMU object.

To remove a PCI device from SMMU objects, call this function with the *sop* argument set to NULL.

Returns:

-1

Failure: *errno* is set.

EBUSY

Failure when attempting to add to a SMMU object a DMA device that is owned by another `smmuman` client.

ENOENT

Failure: when attempting to:

- add a device – the `smmuman` service doesn't know which IOMMU/SMMU unit controls the device being added
- remove a device – the `smmuman` client making the call doesn't own the SMMU object to which the device is attached

0

Success

+1

Success, but the client must call *smmu_mapping_add()* to reissue the memory mappings for this object (see "*Preferred sequence for adding memory mappings and devices*" for more information about why memory mappings may need to be reissued).

smmu_device_report_reserved()

Report which memory region or regions are reserved for a DMA device that requires device-specific reserved memory

Synopsis:

```
#include <smmu.h>

unsigned smmu_device_report_reserved(const union smmu_devid *devid,
                                     unsigned offset,
                                     struct smmu_map_entry *resv,
                                     unsigned nresv);
```

Arguments:

devid

A pointer to the DMA device's ID.

offset

The starting index of the reserved memory array for which information will be returned.

If you set this argument to a location beyond the last reserved entry, you can call *smmu_device_report_reserved()* repeatedly with an ever-increasing offset, retrieving a fixed number of entries at each call, until you have retrieved all the entries.

resv

A pointer to a structure that the function populates with information about the memory regions reserved for the DMA device (see `smmu_map_entry`).

nresv

The number of elements in the array referenced by *resv*; should be greater than `0` (zero).

Library:

libsmmu.a

Description:

The *smmu_device_report_reserved()* function fills in the `smmu_map_entry` data structure referenced by *resv* with information about the reserved memory regions for the device referenced by *devid*.

For more information about reserved memory regions, see the `reserved` option under "*Global options*".

Returns:

-1

Failure: *errno* is set.

0

The *offset* argument is to a location beyond the last reserved entry.

>0

The number of elements filled into the *resv* array.

If there are additional entries to be returned, this returned value will be greater than the value of *nresv*. However, the number of array entries actually filled in is only ever *nresv*.

smmu_fini()

Terminate the client's connection with `smmuman`

Synopsis:

```
#include <smmu.h>

void smmu_fini(void);
```

Library:

libsmmu.a

Description:

The *smmu_fini()* function terminates a client's connection with the `smmuman` service.

WARNING: Before calling *smmu_fini()*, you must call *smmu_obj_destroy()* to destroy the SMMU object(s) to which your client has attached devices.

---

smmu_init()

Initialize contact between the client process and `smmuman`

Synopsis:

```
#include <smmu.h>

int smmu_init(unsigned flags);
```

Arguments:

flags

Must be `0` (zero).

Library:

libsmmu.a

Description:

The *smmu_init()* function checks that the `smmuman` service is running, and connects the client to the service.

To use *smmu_init()* your `smmuman` client must have the SMMU_ABILITY_ATTACH_NAME (`"smmu/attach"`) custom ability before calling the function.

If the client will use the SMF_TARGET flag when calling *smmu_mapping_add()* it must also have the SMMU_ABILITY_TARGET_NAME (`"smmu/target"`) custom ability before calling *smmu_init()*.

Use *procmgr_ability_lookup()* to obtain these custom abilities, and keep the returned ability identifiers in your list of allowed abilities. See *procmgr_ability_lookup()* and *procmgr_ability()* in the *C Library Reference*.

Returns:

-1

Failure: *errno* is set.

0

Success: a connection with the service has been established.

smmu_mapping_add()

Add a memory mapping to or remove a memory mapping from a SMMU object

Synopsis:

```
#include <smmu.h>

int smmu_mapping_add(struct smmu_object *sop,
                     unsigned flags,
                     pid_t pid,
                     unsigned num_entries,
                     const struct smmu_map_entry *entries,
                     _Uint64t target);
```

Arguments:

sop

A pointer to the SMMU object to be modified with new mappings.

flags

Flags that determine the permissions of memory regions mapped to SMMU objects (see `smmu_mapping_flags`).

pid

For future use.

num_entries

The number of mappings to add to or remove from the SMMU object.

entries

An array with the mapping entries to add to or remove from the SMMU object.

target

Set to `0` (zero). Ignored unless SMF_TARGET is set. Currently only the QNX Hypervisor should set SMF_TARGET (see `smmu_mapping_flags`).

Library:

libsmmu.a

Description:

The *smmu_mapping_add()* function adds or removes memory regions that DMA devices are allowed to access. Any DMA device attached to the SMMU object modified by this function will get access to the newly added memory region or regions, in addition to the memory regions already specified in the startup configuration (see "*Configuring smmuman*" in the "*smmuman*" chapter).

To remove memory region mappings, call *smmu_mapping_add()* without specifying any of the SMF_READ, SMF_WRITE, or SMF_EXEC permissions for the *flags* argument (i.e., the *flags* argument set to SMF_NONE; see *smmu_mapping_flags*).

The `smmuman` service doesn't impose page alignment for the starting address and the length of the allowed memory region. Such restrictions are dictated by the hardware requirements and implemented in the architecture-specific and board-specific support libraries (**smmu-*.so**).

With the currently supported IOMMU/SMMUs, these libraries round the starting address *down* and the length *up* to 4K page boundaries. Future IOMMU/SMMU hardware support code may implement support for different hardware requirements.

Returns:

-1

Failure: *errno* is set.

0

Success.

smmu_obj_create()

Create a SMMU object to which devices and their memory mappings can be added

Synopsis:

```
#include <smmu.h>

struct smmu_object *smmu_obj_create(unsigned flags);
```

Arguments:

flags

Always set this argument to SOCF_NONE (see "*smmu_obj_create_flags*").

Library:

libsmmu.a

Description:

The *smmu_obj_create()* function creates a new SMMU object, to which the `smmuman` service can map memory regions with their access permissions, and attach devices.

All devices attached to a SMMU object share the same memory mappings and access permissions. Whenever you call a *smmu_device_add_*()* function to add a new device to a SMMU object, that device automatically receives the memory mappings and access permissions specified for that object.

Similarly, if you call *smmu_mapping_add()* to modify the memory mappings and/or access permissions associated with a SMMU object, your modifications will apply to all the devices attached to that object.

Thus, if your devices require access to different memory regions, or different access permissions to the same region or regions (e.g, one device needs to read, the other needs to write), you must create a SMMU object for each set of access permissions, then attach your devices to the appropriate objects.

Call *smmu_obj_destroy()* to destroy a SMMU object when you don't need it any more.

Destroy all SMMU objects you have created before calling *smmu_fini()* to terminate your connection to the `smmuman` service.

Returns:

NULL

Failure: *errno* is set.

non-NULL

Success: returns a pointer to the newly created SMMU object (`smmu_object`).

smmu_obj_destroy()

Destroy a SMMU object

Synopsis:

```
#include <smmu.h>

void smmu_obj_destroy(struct smmu_object *sop);
```

Arguments:

sop

A pointer to the `smmu_object` structure for the SMMU object to be destroyed.

Library:

libsmmu.a

Description:

Destroy the SMMU object referenced by *sop*.

CAUTION:

Always call *smmu_obj_destroy()* to destroy all SMMU objects you have created before calling *smmu_fini()* to terminate your connection to the `smmuman` service.

Returns:

n/a

smmu_safety()

Query the `smmuman` *service for its safety status*

Synopsis:

```
#include <smmu.h>

int smmu_safety(void);
```

Arguments:

None.

Library:

libsmmu.a

Description:

The *smmu_safety()* function checks the `smmuman` service's safety status. See *safety* in "*Global options*" for information about configuring your system's response to the presence of components that aren't safety-certified.

WARNING: If you are building a safety-related system, you must use safety-certified component. For more information, see "*Component safety-certification status*" in the "*smmuman*" chapter.

Returns:

SMMUMAN_SAFETY_ERROR

Failure: *errno* is set. The function was unable to retrieve the safety status of the `smmuman` service.

SMMUMAN_SAFETY_NO

The `smmuman` service or some dependency (e.g., `procnto`) is *not* a safety-certified component.

SMMUMAN_SAFETY_YES

The `smmuman` service and all its dependencies *are* safety-certified components.

smmu_xfer_notify()

Register to be notified of DMA device memory access transgressions

Synopsis:

```
#include <smmu.h>

int smmu_xfer_notify(const struct sigevent *evp);
```

Arguments:

evp

A pointer to the `sigevent` data structure.

Library:

libsmmu.a

Description:

Clients of the `smmuman` service can use the *smmu_xfer_notify()* function to register to have the service inform them when a DMA device without the appropriate permissions attempts to access memory.

When the `smmuman` service learns that an IOMMU/SMMU has refused a DMA device's memory access attempt, the service delivers a `sigevent` to the client that registered with *smmu_xfer_notify()* to inform it that a transgression record is available. The client can then call *smmu_xfer_status()* to retrieve the record with the information about the DMA device transgression.

The `smmuman` client registering for and receiving updates doesn't have to be the client that owns the SMMU objects and has added devices and mapped memory regions to it. You could have some clients (e.g., device drivers) create objects, add devices and map memory regions, and another client (i.e., some other process in your system) monitor transgressions.

Ensuring that your client receives notifications

The `smmuman` service doesn't internally queue up notifications from hardware of memory access transgressions by DMA devices; it doesn't save notifications and deliver them later to clients when they register for notifications.

This means that if a client isn't attached to the `smmuman` service and registered for notifications when a DMA device attempts an illegal memory access, the client will never know about the attempted transgression.

Note also that when the `smmuman` service informs a client that an IOMMU/SMMU has refused a DMA device's memory access attempt, it clears that client's registration for notification of DMA device memory access transgressions. The client must call *smmu_xfer_notify()* to re-register for notification.

Returns:

-1

Failure: *errno* is set.

>0

Success.

smmu_xfer_status()

Retrieve the record of a DMA device's attempt to access memory outside its permitted region or regions

Synopsis:

```
#include <smmu.h>

int smmu_xfer_status(struct smmu_status *ssp);
```

Arguments:

ssp

A pointer to the `smmu_status` structure to be populated with the error information.

Library:

libsmmu.a

Description:

The *smmu_xfer_status()* function retrieves one status record of a DMA device memory access transgression.

If your `smmuman` client needs to be informed of DMA device memory access transgressions, after it connects to the service, the client should call *smmu_xfer_notify()* to register to receive notifications of remapping errors.

On receipt of a notification, the client can call *smmu_xfer_status()* to retrieve the record of the transgression and decide what to do about it.

The `smmuman` service can store up to 64 records of attempted transgressions per client attached to the service. If the DMA device produces more than 64 transgressions before a call to *smmu_xfer_status()*, the `smmuman` begins dropping new transgression records, and sets the SSF_DROPPED_FAULTS bit in the `smmu_status`'s `flag` member.

You can check this bit to learn if the `smmuman` service had to drop one or transgression records before your client was able to read them.

Retrieving a records clears the buffer space used by that record.

When the `smmuman` service informs a client that an IOMMU/SMMU has refused a DMA device's memory access attempt, it clears that client's registration for notification of DMA device memory access transgressions. The client must call *smmu_xfer_notify()* to re-register for notification.

The `smmuman` service can only pass on information that has been supplied by the hardware. For example, if the hardware supplies the page offset portion of the address a DMA device attempted to access but to which it was refused access, a call to *smmu_xfer_status()* can retrieve that information. However, if the hardware doesn't make that information available, it won't be available to the `smmuman` service or to its clients.

If you will do more than simply log a DMA device's attempted transgression, then you should read the device registers directly to retrieve the information you need.

Returns:

-1

Failure: *errno* is set.

1

Success: a record was found and the `smmu_status` structure was populated with information about the transgression.

0

Success, but no record of a transgression was found.

Appendix A
Example program

Example of a program that connects to the `smmuman` service and monitors IOMMU/SMMU transgressions reported by the service.

The following program is taken from a system that maps all DMA devices in the `smmuman` startup configuration (see "*Startup mappings*"). It monitors transgressions reported by the `smmuman` service, but doesn't create SMMU objects, or memory mappings:

```
#ifdef __USAGE
%C - SMMU violations monitor

Syntax:
%C

Options:
    -v verbosity
#endif

#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
#include <smmu.h>
#include <inttypes.h>
#include <sys/neutrino.h>
#include <sys/siginfo.h>
#include <sys/slogcodes.h>
#include <sys/procmgr.h>

#define PULSE_CODE_SMMU_VIOLATION 0
#define _SLOGC_SMMUMON (_SLOGC_PRIVATE_END - 6)

static int verbosity_level = 0;

static int smmumon_slogf(int severity, int verbosity, int vlevel, const char *fmt, ...)
{
    int     ret;
    va_list arglist;
```

```
    ret = 0;

    if (severity <= _SLOG_ERROR || verbosity > 5) {
        va_start(arglist, fmt);
        vfprintf(stderr, fmt, arglist);
        va_end(arglist);
        fprintf(stderr, "\n");
    }
    if (vlevel <= 4 || verbosity >= vlevel) {
        va_start(arglist, fmt);
        ret = vslogf(_SLOGC_SMMUMON, severity, fmt, arglist);
        va_end(arglist);
    }
    return (ret);
}

static void options(int argc, char **argv)
{
    int     opt;
    while (optind < argc)
    {
        while ((opt = getopt(argc, argv, "v")) != -1)
        {
            switch (opt)
            {
                case 'v':
                    if (verbosity_level < 20) {
                        verbosity_level++;
                    }
                    break;
                default:
                    smmumon_slogf(
                        _SLOG_ERROR, 1, 0,
                        "options: unexpected parameter\n");
                    exit(-1);
            }
        }
    }
}

static int process_all_violations(void)
{
    int result;
```

```
    for(;;) {
        struct smmu_status status;
        result = smmu_xfer_status(&status);
        if (result == -1) {
            smmumon_slogf(
                _SLOG_ERROR, verbosity_level, 0,
                "process_all_violations: smmu_xfer_status failed, errno=%d\n",
                errno);
            return -1;
        }

        if (result == 0)
        {
            return 0;
        }

        if (result != 1) {
            smmumon_slogf(
                _SLOG_ERROR, verbosity_level, 0,
                "process_all_violations: unexpected result from smmu_xfer_status,"
                " result=%d\n", result);
            return -1;
        }

        switch(status.devid.type) {
            case SDT_PCI: {
                smmumon_slogf(
                    _SLOG_CRITICAL, verbosity_level, 0,
                    "pci device violation detected : "
                    "fault_addr=0x%" PRIx64 " "
                    "hw_specific=0x%" PRIx64 " "
                    "flags=0x%x "
                    "pci bdf=%d:%d:%d"
                    "\n",
                    status.fault_addr,
                    status.hw_specific,
                    status.flags,
                    status.devid.pci.bus,
                    status.devid.pci.dev,
                    status.devid.pci.func);
                break;
            }
            case SDT_MMIO: {
```

```
                smmumon_slogf(
                    _SLOG_CRITICAL, verbosity_level, 0,
                    "mmio device violation detected : "
                    "fault_addr=0x%" PRIx64 " "
                    "hw_specific=0x%" PRIx64 " "
                    "flags=0x%x "
                    "mmio.phys=0x%" PRIx64 " "
                    "mmio.length=0x%x"
                    "\n",
                    status.fault_addr,
                    status.hw_specific,
                    status.flags,
                    status.devid.mmio.phys,
                    status.devid.mmio.length);
                break;
            }
            default:
            {
                smmumon_slogf(
                    _SLOG_CRITICAL, verbosity_level, 0,
                    "unknown device type violation detected, "
                    "type=%d : "
                    "fault_addr=0x%" PRIx64 " "
                    "hw_specific=0x%" PRIx64 " "
                    "flags=0x%x "
                    "\n",
                    status.devid.type,
                    status.fault_addr,
                    status.hw_specific,
                    status.flags);
                break;
            }
        }
    }
    return -1;
}

static int wait_for_violations(int channel, int coid)
{
    int result;

    struct sigevent event;
    SIGEV_PULSE_INIT(&event, coid, -1, PULSE_CODE_SMMU_VIOLATION, NULL);
```

```
    result = smmu_xfer_notify(&event);
    if (result!=0) {
        smmumon_slogf(
            _SLOG_ERROR, verbosity_level, 0,
            "wait_for_violations: smmu_xfer_notify-1 failed , errno=%d\n",
            errno);
        return -1;
    }

    for(;;) {
        struct _pulse pulse;
        result = MsgReceivePulse(channel, &pulse, sizeof(pulse), NULL);
        if (result != 0) {
            smmumon_slogf(
                _SLOG_ERROR, verbosity_level, 0,
                "wait_for_violations: MsgReceivePulse failed, errno=%d\n",
                errno);
            return -1;
        }

        if (pulse.code != PULSE_CODE_SMMU_VIOLATION) {
            smmumon_slogf(
                _SLOG_ERROR, verbosity_level, 0,
                "wait_for_violations: unexpected pulse.code=%d\n",
                pulse.code);
            return -1;
        }

        result = process_all_violations();
        if (result != 0) {
            smmumon_slogf(
                _SLOG_ERROR, verbosity_level, 0,
                "wait_for_violations: process_all_violations failed, errno=%d\n",
                errno);
            return -1;
        }

        result = smmu_xfer_notify(&event);
        if (result!=0) {
            smmumon_slogf(
                _SLOG_ERROR, verbosity_level, 0,
                "wait_for_violations: smmu_xfer_notify-2 failed , errno=%d\n",
```

```
                errno);
            return -1;
        }
    }

    return -1;
}

static int connect_attach_channel_and_process(int channel)
{
    int coid;
    coid = ConnectAttach_r(0, 0, channel, _NTO_SIDE_CHANNEL, 0);
    if(coid < 0) {
        smmumon_slogf(
            _SLOG_ERROR, verbosity_level, 0,
            "connect_attach_channel_and_process : ConnectAttach_r %d\n",
            -coid);
        return -1;
    }

    int result;
    result = wait_for_violations(channel,coid);

    ConnectDetach_r(coid);

    return result;
}

static int open_channel_and_process(void)
{
    int channel;
    channel = ChannelCreate_r(0);
    if(channel < 0) {
        smmumon_slogf(
            _SLOG_ERROR, verbosity_level, 0,
            "open_channel_and_process: ChannelCreate_r failed: %d\n",
            -channel);
        return -1;
    }

    int result;
    result = connect_attach_channel_and_process(channel);
```

```
    ChannelDestroy_r(channel);

    return result;
}

int main(int argc, char *argv[])
{
    int result;

    options(argc,argv);

    result = procmgr_daemon( 0,
        PROCMGR_DAEMON_NODEVNULL | PROCMGR_DAEMON_NOCLOSE);

    if (result < 0) {
        smmumon_slogf(
            _SLOG_ERROR, verbosity_level, 0,
            "main: procmgr_daemon failed : %s\n", strerror(errno));
        return -1;
    }

    result = smmu_init(0);
    if (result != 0) {
        smmumon_slogf(
            _SLOG_ERROR, verbosity_level, 0,
            "main: smmu_init failed, errno=%d\n",errno);
        return -1;
    }

    result = open_channel_and_process();

    smmu_fini();

    return result;
}
```

Appendix B
Terminology

The following terms are used throughout the SMMUMAN documentation.

DMA

Direct Memory Access

Guest

A *guest* is an OS running in a QNX Hypervisor `qvm` process; this process presents the virtual machine (VM) in which the guest runs.

Guest-physical address

A memory address in guest-physcial memory (see "Guest-physical memory" below).

Guest-physical memory

The memory assembled for a VM by the `qvm` process that creates and configures the VM. ARM calls this assembled memory *intermediate physical memory*; Intel calls it *guest physical memory*. For simplicity, regardless of the platform, we will use the term Bugnion, Nieh and Tsafrir use in *Hardware and Software Support for Virtualization* (Morgan & Claypool, 2017): "guest-physical memory", and the corresponding term: "guest-physical address".

Host

Either the *development host* (the desktop or laptop, which you can connect to your target system to load a new image or debug), or the *hypervisor host domain*.

Host-physical address

A memory address in host-physcial memory (see "Host-physical memory" below).

Host-physical memory

The physical memory; this is the memory seen by the hypervisor host, or any other entity running in a non-virtualized environment. (see "*Guest-physical memory*" above).

Hypervisor

A microkernel that includes virtualization extensions. In a QNX environment, these extensions are enabled by adding the `module=qvm` directive in a QNX buildfile.

IOMMU

Input/Output Memory Management Unit. A memory management unit (MMU) that connects a DMA–capable I/O bus to the main memory. Like a traditional MMU, which translates CPU-visible intermediate addresses to physical addresses, an IOMMU maps device-visible intermediate addresses (also called device addresses or I/O addresses in this context) to physical addresses. This mapping ensures that DMA devices cannot interact with memory outside their bounded areas.

IPMMU

Intellectual Property MMU. IOMMU on Renesas R-Car platforms.

QNX Hypervisor

The running instance of a QNX hypervisor.

Reserved memory region

A *reserved memory region* is a region in memory required by the specification for a particular device, typically to hold some information such as tables the device needs in order to operate. Unless you specify otherwise, the `smmuman` service looks after these regions (see *smmu_obj_create_flags* in the "*SMMUMAN Client API Reference*" chapter).

A reserved memory region is *not* the same as a memory area you map to a SMMU object.

SMMU

System Memory Management Unit. An ARM implementation of an IOMMU.

VT-d

Intel Virtualization Technology for Directed I/O (VT-d) is an Intel implementation of an IOMMU.

Index