Document QMS3289

QNX OS for Safety
Safety Manual

Version 9.0

Document Number	QMS3289
Version	9.0
Project ID	QOS222
Document Generated	June 29, 2022
Document Status	Approved

June 29, 2022

Table of Contents

CHAPTER 1

Introduction

Contents

1.1 This document

1.1.1 Document identification

This document is identified by the unique number QMS3289 together with the version number 9.0.

1.1.2 Document purpose

This document accompanies the *QNX OS for Safety* and describes the constraints within which that product must be used to ensure that its certification is not invalidated. The application developer must read and understand the contents of this document and the limits it places on an application, before incorporating the *QNX OS for Safety* into a product.

The *QNX OS for Safety* is, under the definitions given in ISO 26262, a Safety Element out of Context (SEooC) and is also compliant to IEC 61508 and IEC 62304.

These standards place certain requirements on the information contained in this document:

- Chapter 2 provides an overview of the product scope and expected development environment for target systems.

- Chapter 3 provides restrictions and recommendations related to correctly setting up a development environment.
- Chapter 4 provides restrictions and recommendations related to application development.
- Chapter 6 provides restrictions and recommendations related to security features that are used to configure aspects of freedom from interference.
- Chapter 7 provides restrictions and recommendations related to handling failures in QOS itself at the system level.
- Chapter 8 provides restrictions and recommendations related to performance analysis and tuning.

1.2 Document audience

Use this manual if you are responsible for system development, and you intend to use the *QNX OS for Safety* as a component within your system.

This document is specifically aimed at the following staff:

1. system architects
2. application designers and programmers
3. safety assessors (either independent or within the application development team)
4. other members of the functional safety management team

Before deploying the *QNX OS for Safety* within a certified system or system to be certified, technical personnel are expected to have performed the following:

1. read and understood the safety-related concepts, application prerequisites and responsibilities presented throughout this document in the context of the safety standard to which they are working
2. ensured that the *QNX OS for Safety* has been installed and is used in accordance with the associated installation guide
3. considered the (optional) training available from BlackBerry QNX for application developers and other users of this safety manual, and otherwise ensure appropriate competency for safety-related application development, i.e., users are fully qualified, specially trained, and experienced in safety-system development

1.3 Applicability

The *QNX OS for Safety* (hereafter referred to as the “QOS”) is a general-purpose component and is based on the QNX Neutrino kernel, which is used in thousands of diverse applications worldwide.

This component has been developed in accordance with the requirements for a SEooC as defined in ISO 26262, and the requirements for a compliant item as defined in IEC 61508 and IEC 62304, and has been independently assessed and certified against those standards. An important part of the software life cycle used to develop and maintain the QOS is concerned with Verification and Validation (V&V) so that every release of this component is subjected to rigorous certified quality and safety assurance processes.

BlackBerry QNX's commitment to V&V rigor doesn't absolve system developers from performing V&V on the components they use in their safety-related applications. The application may seek to use the QOS in a new or unproven way, and it is therefore the responsibility of the system developer to perform the V&V needed to support their application's safety compliance case.

1.4 Related documentation

The QNX Neutrino RTOS, on which the QOS is based, has a rich set of documentation, and information in those documents can be considered applicable to the QOS unless specifically stated otherwise in this document.

In particular, the following documents will be of interest to the readers of this Safety Manual:

1. *Installing and Using the QNX OS for Safety 2.2.2*. This is available from

   `http://www.qnx.com/download/feature.html?programid=49542`

2. The *QNX Software Development Platform System Architecture Guide.* This is available from

   `http://www.qnx.com/developers/docs/7.1/#com.qnx.doc.neutrino.sys_arch/topic/about.html`

   and provides details about both the design philosophy of the QNX Neutrino RTOS and the architecture used to implement it. The following are specific sections (or chapters) in the System Architecture guide that are relevant to the *QNX OS for Safety*:

   - About This Guide
   - The Philosophy of QNX Neutrino
   - The QNX Neutrino Microkernel
   - Interprocess Communication (IPC)
   - The Instrumented Microkernel
   - Multicore Processing
   - Process Manager
   - Dynamic Linking

- Resource Managers
- Adaptive Partitioning
- Glossary

3. The *QNX Software Development Platform C Library Reference.* This is available from

   http://www.qnx.com/developers/docs/7.1#com.qnx.doc.neutrino.lib_ref/topic/about.html

   and describes the C functions, data types, and protocols that are included as part of the QNX Neutrino RTOS. The following are specific sections (or chapters) that are relevant to the product:

   - About This Reference
   - What's in a Function Description?
   - Full Safety Information

4. The *QNX Software Development Platform Adaptive Partitioning User's Guide.* This is available from

   http://www.qnx.com/developers/docs/7.1/com.qnx.doc.adaptivepartitioning.userguide/topic/about_howtouseguide_.html

   and describes the Adaptive Partitioning Scheduler that may be used with the QOS.

For a detailed overview of the BlackBerry QNX documentation, see the *QNX Software Development Platform* guide which is available from

http://www.qnx.com/developers/docs/7.1/#com.qnx.doc.qnxsdp.nav/topic/bookset.html

1.5 Terminology

In the remainder of this document, the acronym QOS is used to refer to the *QNX OS for Safety*.

When this document refers to ISO 26262, unless otherwise specified, it should be taken as a reference to ISO 26262:2018. When this document refers to IEC 61508, unless otherwise specified, it should be taken as a reference to IEC 61508:2010. When this document refers to IEC 62304, unless otherwise specified, it should be taken as a reference to IEC 62304:2015 edition 1.1. Other acronyms are listed in table 1.1.

The key words “MUST", “MUST NOT", “REQUIRED", “SHALL", “SHALL NOT", “SHOULD", “SHOULD NOT", “RECOMMENDED", “MAY", and “OPTIONAL" used within this document are to be interpreted as described in RFC 2119, as published by the Internet Engineering Task Force (IETF) and clarified in RFC8174, and are available at http://www.rfc-editor.org/rfc/rfc2119.txt.

Table 1.1: Acronyms used in this document

Acronym/Term	Meaning	Notes
(Software) Application	A software program that provides a useful function in the context of the wider system. In a client-server architecture, the applications are the clients at the highest level in the software system.	
Application Processor	A processor designed to provide general purpose set of processing capabilities with a statistically high instruction throughput. The capabilities of modern application processors include integer and floating-point arithmetic along with simultaneous multiple data operations (vector operations), as well as the use of memory virtualisation for software process isolation. In comparison, microcontrollers and digital signal processors (DSP) are classes of processor with different operational goals and specifications.	
API	Application Programming Interface	
APS	Adaptive Partitioning Scheduler	
ASIL	Automotive Safety Integrity Level	See § 3.6 of ISO 26262-1:2018
BSP	Board Support Package	
Core, Processor core	A structural grouping (or integration) of the processor's functional elements that can be reused and aggregated. A CPU is one of these functional elements, but there are others, such as the FPU, level 1 cache, bus interfaces, etc. Therefore, CPU, processor, and processor core (or just "core" when the context of use is clear) are not strictly interchangeable terms.	
Critical Process	A process that will cause the system to crash upon termination	

CPU	Central Processing Unit - the heart of a processor providing the logic for machine instruction execution. The CPU in a modern application processor is just one of many elements comprising a processor core. The modern term "core" therefore supersedes its use, except for a few legacy usages relating to configuration entries.	
DSS	Design Safe State	
ECC memory	Error-correcting code memory	
FPU	Floating Point Unit - an instruction execution unit dedicated to floating point arithmetic operations.	
gid	Group identifier, used to uniquely identify a group	
HAM	High-Availability Manager	
HRT	High Resolution Timer	
IDA	Instant Device Activation	
IFS	Image File System	
IPC	Interprocess Communication	
IPI	Inter-Processor Interrupts	
ISR	Interrupt Service Routine	
libc	C Library	
malicious code	Code that gives unauthorized access to the attacked system and then exposes sensitive information	
Multicore	An *application* processor with more than one processing core that can provide concurrent execution of collaborative threads of software execution. An essential requirement for achieving collaborative concurrency is a coherent memory sub-system. See Symmetric multiprocessing (SMP). Not to be confused with multiple processors sharing a common bus and main memory in a System on Chip (SoC), but lacking hardware-based memory coherency and potentially also being with different processor specifications.	
NMI	Non-maskable interrupt	
PR	Problem Report	
procnto	The microkernel and process manager	

Process, Software Process	A running program. A process owns resources, including memory, and software execution threads managed by the operating system.	
QHS	QNX Hypervisor for Safety	
QM	Quality Management	
Safety Function	A function with a safety impact	
Safety Application	An application that implements a safety function	
Safe System	A system developed with safety as its highest priority	
SDP	QNX Software Development Platform	
SEooC	Safety Element out of Context	See § 9 of ISO 26262-10:2018
SIL	Safety Integrity Level	See § 7.6.2.9 of IEC 61508-1
SMP	Symmetrical Multi-Processing - An actively managed form of parallel software execution whereby two or more processor cores with access to a shared and coherent memory sub-system are given work to perform by an executive operating system. An SMP operating system can make use of all of the cores under its control freely because they are required to provide an identical instruction execution capability, that is they are homogeneous in their instruction execution characteristics. See also "Multicore"	
SoC	System on Chip - an integration of many processor, memory, and peripheral elements into a single packaged semiconductor device.	
syspage	System page contains essential information about the system.	
TCL	Tool Confidence Level	
TSC	Time-stamp Counters	
Über-Authorities	A Über-Authority takes control of the processor core(s) in a system for a certain duration. The operating system is not aware of this.	
uid	User identifier, used to uniquely identify a user	
V&V	Verification and Validation	

Restrictions in this document are written with the following formatting:

RST-0000. The component SHALL...

Recommendations in this document are written with the following formatting:

REC-0000. The component SHOULD...

CHAPTER 2

QOS Product Overview

Contents

This chapter defines the environment within which it is assumed that the QOS will be installed. It contains:

Section *2.2* An informal description of the environment within which the QOS is designed to be used.

Section *2.3* The boundary of the QOS.

Section *2.4* The types of processor on which the QOS may be executed.

Section *2.5* The variants of the QOS.

Section *2.7* The toolchain associated with the QOS.

Section *2.8* The organizational environment during development—the responsibilities that fall on the user and those that fall onto the developer of the QOS.

2.1 Safety goal

The QOS, when used in accordance with the constraints given in this document, has been designed to meet the requirements of IEC 61508, ISO 26262, and IEC 62304 for a component to be used in items in order to realize safety goals up to classification SIL-3 (IEC 61508), ASIL-D (ISO 26262), and Class C (IEC 62304).

2.2 Informal description of the environment

Figure 2.1 is adapted from figure 22 in ISO 26262-10:2018 and illustrates the manner in which the QOS, as a SEooC, should be integrated into an ISO 26262 “item”. In particular, it is the responsibility of the user of the QOS to ensure that the assumptions made by BlackBerry QNX of the use to which the QOS will be put match those required by the Item into which the QOS is to be integrated.

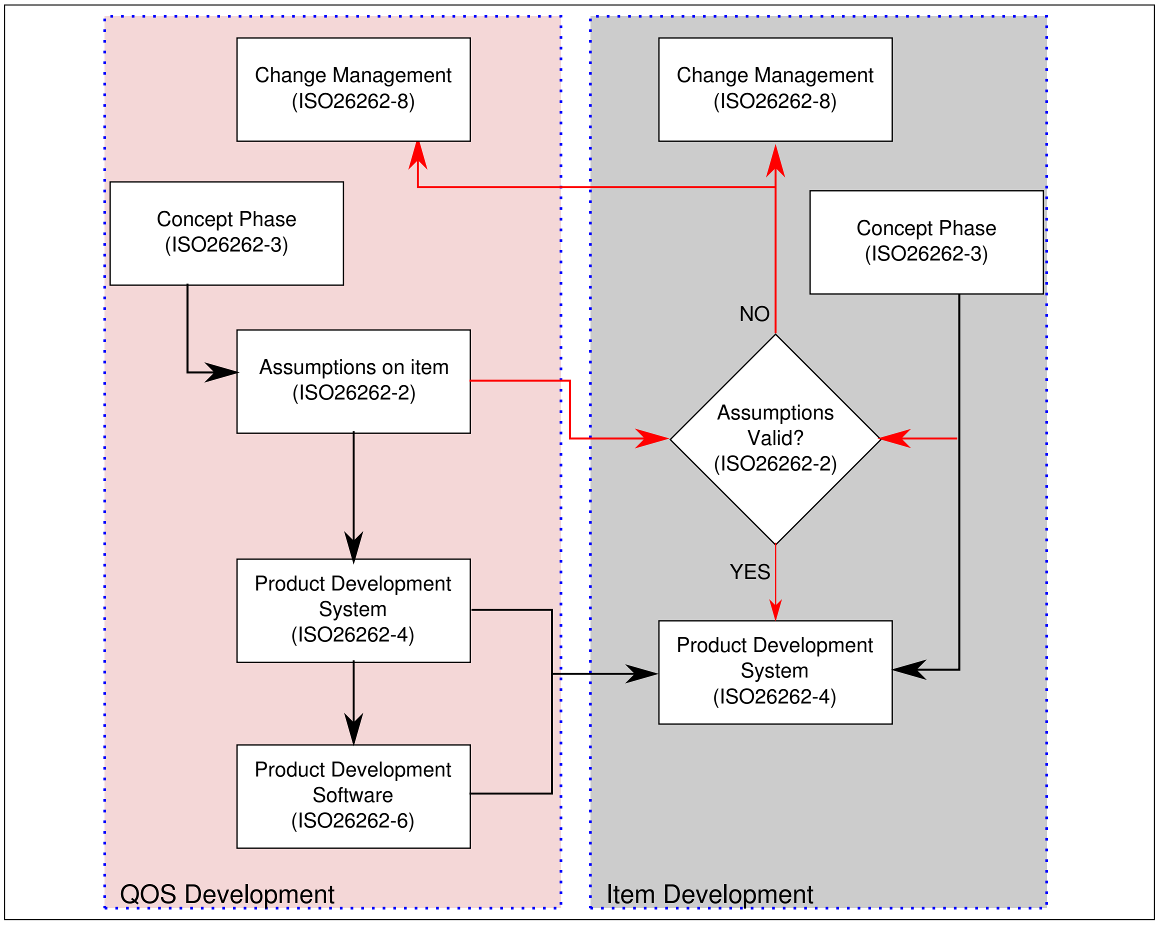
Change Management (ISO26262-8)
Change Management (ISO26262-8)
Concept Phase (ISO26262-3)
Concept Phase (ISO26262-3)
NO
Assumptions on item (ISO26262-2)
Assumptions Valid? (ISO26262-2)
YES
Product Development System (ISO26262-4)
Product Development System (ISO26262-4)
Product Development Software (ISO26262-6)
QOS Development
Item Development

Figure 2.1: Application of the QOS

As the QOS is a software-only product, except as listed in later sections, the responsibility for detecting and handling failures of the hardware environment in which it executes (processor, memory, power supply, etc.) falls outside the QOS.

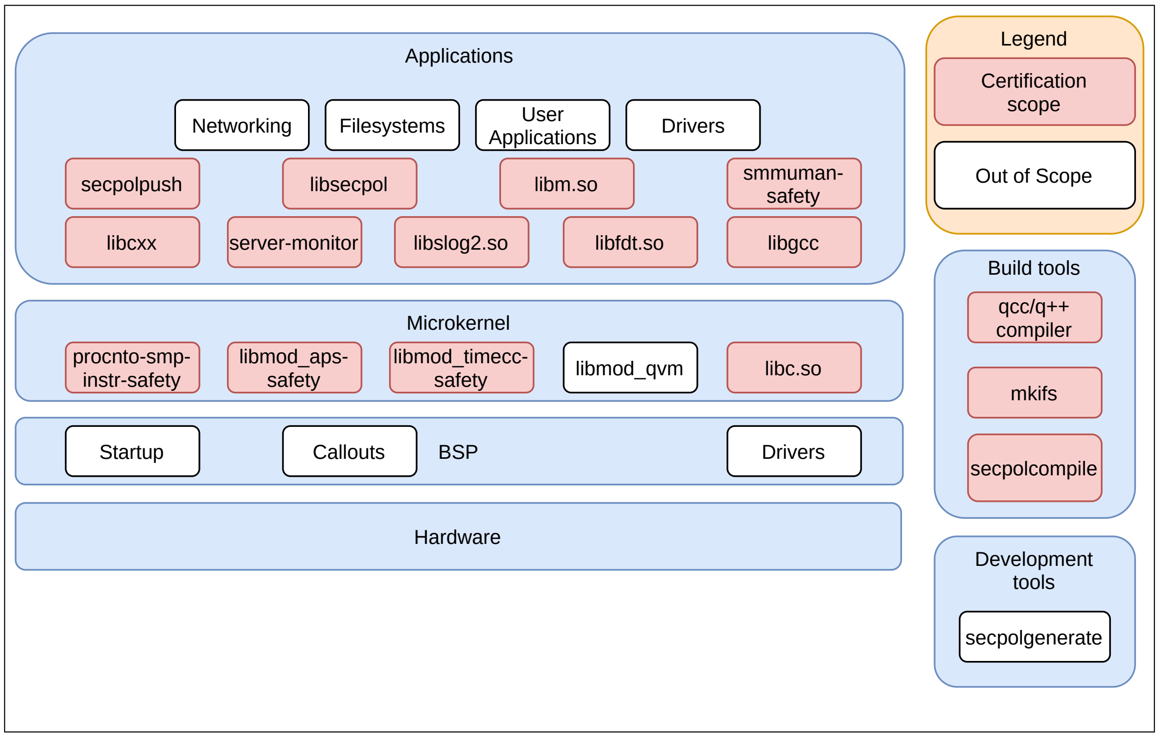

Figure 2.2: Boundary of the QOS

2.3 Boundary of the certified product

Figure 2.2 informally illustrates the scope of the QOS. Note that the list of certified binaries should be consulted for the complete list of binaries and tools (see Section 3.3.2 for details). Note in particular that the figure depicts networking, device drivers, and filesystems (which normally form part of a monolithic operating system) as lying outside the boundary of the QOS and being part of the environment within which it operates. User applications are peers of these components.

The QOS boundary includes, but is not limited to:

- the QNX Neutrino RTOS C Library interface (`libc`), C++ Library (`libcxx`) [1] Mathematical Library (`libm`), libgcc, Security Policy Library (`libsecpol`), and Run-Time Library (`crt`) interfaces
- the `procnto` Microkernel and the Process Manager
- the Adaptive Partitioning Scheduler (APS) kernel module
- the timecc kernel module
- the SMMU Manager (`smmuman`) and associated support libraries
- generic safety functions required to support safe startup and execution of the QOS
- the `mkifs` utility to generate filesystem images

[1] Note that in contrast to other components, libcxx may be used in order to achieve safety goals up to ASIL B of ISO 26262

- the `secpolgenerate` and the `secpolpush` utilities to generate and load system security policies, respectively
- the development toolchain (e.g., C and C++ compiler)

For a complete list of components included within the scope of certification, please see the revision list for this product. Any components not listed in the revision list should be considered to be outside the scope of certification.

The startup code, which forms part of the BSP, lies outside the QOS. It exists during the startup of the system and, once control is handed to the QOS, it is mapped out of the processor's address space. It therefore does not contribute to unused (or "dead") code in the system.

Note that some C library functions provide an interface to other portions of the environment, which may or may not be within the scope of the QOS. For example, the *open()* function is part of the C library and thus within the scope of the QOS, but it may be used to open a connection to a resource manager that is outside the scope of the QOS.

The QOS is installed as an add-on to the BlackBerry QNX Software Development Platform (SDP) 7.1 installation. Drivers and other executables provided in the SDP 7.1 installation lie outside the scope of the QOS. These components may be used in applications designed to satisfy safety standards, but the justification for their use is the responsibility of the application designer.

2.4 Hardware assumptions

The QOS is a software-only product and executes on the following processor architectures:

- ARMv8 (64-bit), little-endian, with between 1 and 8 cores
- x86_64 (64 bit), with between 1 and 24 cores

As described in restriction 13, it is assumed that the hardware on which the QOS is running will have sufficient mechanisms to detect and, if possible, report hardware errors and failures (e.g., the use of Error-correcting code (ECC) memory) at a sufficient level to satisfy the dependability requirements of the system.

2.5 Kernel variants

Several variants of the QNX Neutrino RTOS are available. The QOS includes the `procnto` variant created with SMP support, instrumentation, and safety support named `procnto-smp-instr-safety`.

Other variants of QNX Neutrino/procnto ARE NOT components of the QOS.

2.6 Building the QOS

The QOS is a set of executables, libraries, header files, and host tools. Users do not build or modify any of these components.

2.7 Toolchain

The C toolchain used to build the components of the QOS (C preprocessor, C compiler, assembler and linker) is implicitly part of the QOS's certification process, and this C toolchain is provided as part of the SDP 7.1 and the QOS packages for compiling applications.

In addition, the system security policy compiler (secpolcompile) and C++ compiler are also provided as part of the SDP 7.1 and the QOS packages.

The tool chain, parts of which are classified as TCL3 in ISO 26262 terminology and T3 in IEC 61508 terminology, complies with the applicable requirements for supporting tools according to ISO 26262-8 and off-line support tools according to IEC 61508 and can be used to compile applications with ISO 26262 or IEC 61508 requirements.

2.8 Development assumptions

Figure 2.3 illustrates the steps that BlackBerry QNX assumes that the designer using the QOS will take in creating a system or component for certification to applicable standards. This figure also defines the boundaries of BlackBerry QNX's responsibility.

The result of the QOS development is a software product. The remaining chapters in this document describe its safe use.

The user of the QOS:

1. installs it onto the development host in accordance with the product documentation, in particular in accordance with the document *Installing and Using the QNX OS for Safety 2.2.2*
2. designs the final application incorporating the QOS and carries out the failure analysis and other procedures as required for applicable standards development
3. implements the design in accordance with the requirements contained in this document
4. integrates the design with the Board Support Package (BSP) and target hardware to produce a complete product

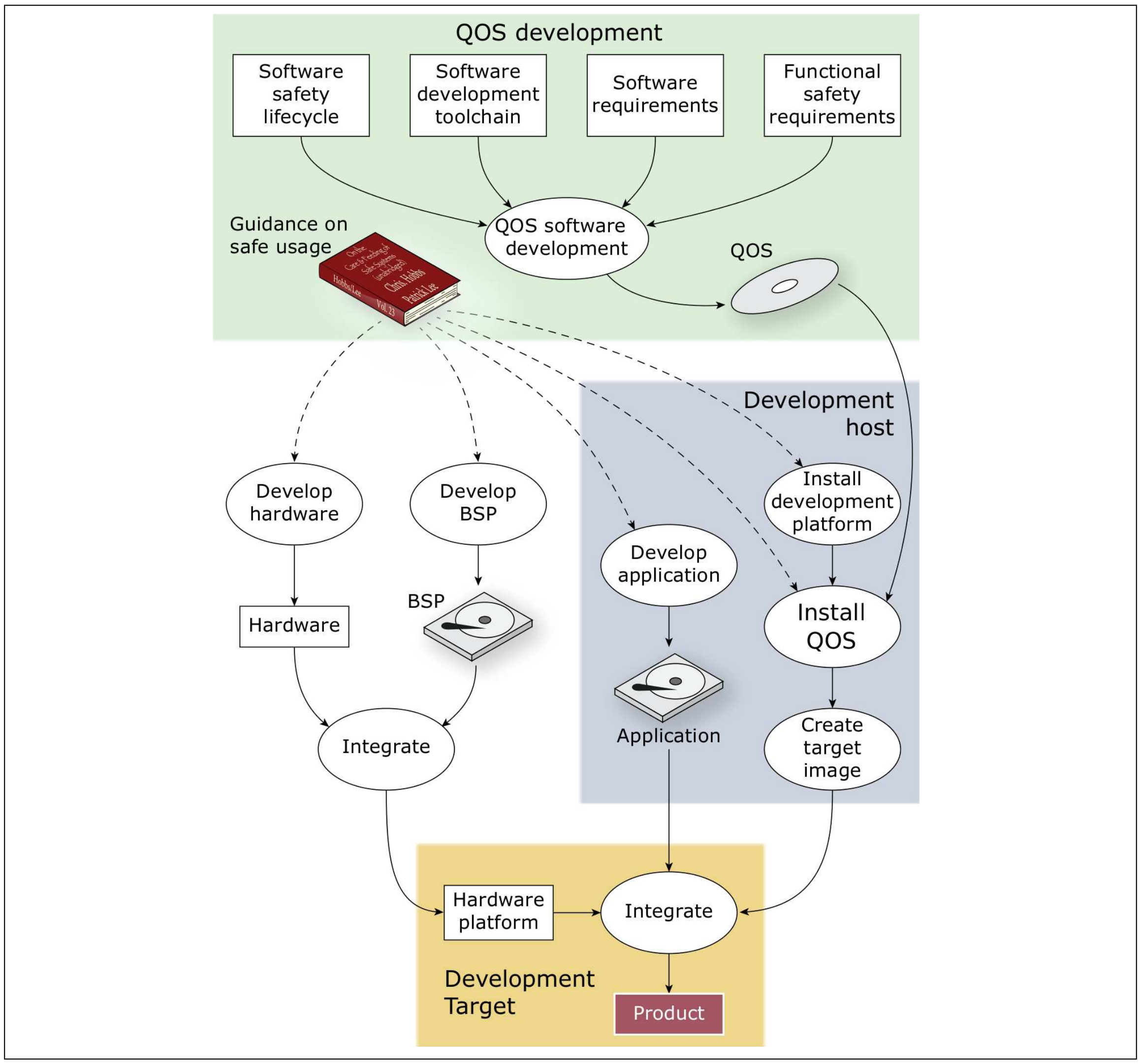

Figure 2.3: System development with QOS

CHAPTER 3

QOS Development Environment

Contents

3.1 Introduction

This chapter lists restrictions and recommended practices related to the installation and configuration of the QOS development environment.

Section 3.2 process for reporting defects

Section 3.3 how to verify the integrity of the QOS installation with software center or manually

Section 3.4 how to check that the development toolchain and system path are properly configured

Section 3.5 criteria for hardware selection

Section 3.6 how to configure the BSP, including the QNX startup module, for a certified system

Section 3.7 hardware configuration for accurate clock measurements

Section 3.8 timecc kernel module for accurate time measurement

Section 3.9 issues related to running a QOS-based system in the presence of *Über-Authorities*, such as a hypervisor, or Intel System Management Mode.

Section 3.10 guidance for consistent system tick

3.2 Reporting product defects

RST-0010. If the user of the QOS detects that the QOS is not behaving in accordance with its published documentation, the user SHALL report that to BlackBerry QNX.

For any product potentially used in safety-critical applications, it is important to be aware of defects. The mechanism for reporting such defects is given in appendix E.3 of this document.

3.3 Verifying the integrity of the installed environment

RST-0003. The QOS SHALL be installed in accordance with the instructions in the *Installing and Using the QNX OS for Safety 2.2.2* guide.

RST-0109. The user SHALL NOT modify any of the following:

- any header files associated with any certified libraries (e.g., `libc`) distributed by BlackBerry QNX, either as part of QOS or as a standalone product.
- preprocessor macros defined by these header files
- preprocessor macros predefined by the compiler (e.g., __QNX__)
- all objects associated (e.g., `__func__`) with reserved identifiers (ISO/IEC 9899:2011 7.1.3).

Header files contain many definitions, including constants, function definitions, and inline functions. Modification to header files and preprocessor macros as listed above could lead to unknown behavior.

During installation of the QOS (in accordance with the document *Installing and Using the QNX OS for Safety 2.2.2*), the integrity of the installer and installed files is automatically confirmed. Each installed file is hashed, the hashes are stored in a manifest file, and the manifest file is cryptographically signed. If any files are corrupted during the installation process, the installation will fail. If the installation completes successfully, the QNX software center has verified that the installed files are correct. However, installed files may become corrupted after installation.

REC-0002. Before building a target image, the user SHOULD check the integrity of all of the components of the QOS.

3.3.1 Verifying environment with QNX Software Center

The QNX Software Center can be used to verify the correct installation of the QOS. To confirm the integrity of installed software, select **Manage Installation** from the **Welcome** screen of the QNX Software Center and go to the **Advanced** tab. First, select the **baseline installation** that includes the **QOS** from the drop-down list at the top right corner. Then, press the **Verify Installation** button. If the installation has become corrupted, the QNX Software Center will report the problem. To view the checksum of each file at the QNX Software Center, go to the **Advanced** tab and right-click on the **columns list**. A pop-up window appears, select **show columns**, and then check the box **Digest SHA512**.

3.3.2 Verifying environment with revision list

As an alternative, the checksum of each file can be verified against the revision list published by the certifying authority. This may be done following installation of the QOS, prior to building a system that uses the QOS, or at run time while a system that uses the QOS is operating. The revision list may be found by searching the certifying authority's web site at `https://fs-products.tuvasi.com/certificates?filter_prod=1&filter_apps=1&keywords=QNX&productcategory_id=1&x=0&y=0` and locating the entry for the current version of the QOS. The checksum is generated using SHA512 with base64 encoding.

For example, the checksum can be generated on Linux and QOS using the `openssl` utility:

```
openssl dgst -binary -sha512 [somefile] | openssl base64 | sed -z 's/\n//'
```

The revision list contains two entries for files that are symbolic links. The first entry contains a LinkSHA512 value and the second contains a TargetSHA512 value. The TargetSHA512 value (contents of the file pointed to) is the value to verify on the command-line. The LinkSHA512 (checksum of the link) can only be verified using the QNX Software Center but is implicitly verified by checking the TargetSHA512 value of the link.

3.4 Checking environment configuration

REC-0061. The system developer SHOULD generate and examine the map files for all the applications to ensure that the binary files are using the correct versions of the QNX Software Development Platform (SDP), and all the dependent libraries.

An example of a map file for a "Hello.c":

```
Memory Configuration

Name             Origin             Length             Attributes
*default*        0x0000000000000000 0xffffffffffffffff
```

```
Linker script and memory map

LOAD ~/qnx710/target/qnx7/x86_64/lib/crt1.o
LOAD ~/qnx710/target/qnx7/x86_64/lib/crti.o
LOAD ~/qnx710/host/linux/x86_64/usr/lib/
      gcc/x86_64-pc-nto-qnx7.1.0/8.3.0/pic/crtbegin.o
LOAD hello.o
LOAD ~/qnx710/target/qnx7/x86_64/lib/libc.so
LOAD ~/qnx710/target/qnx7/x86_64/lib/libcS.a
LOAD ~/qnx710/target/qnx7/x86_64/lib/gcc/8.3.0/libgcc_s.so
```

This QOS version requires the SDP version to be 7.1.0, and library versions can be found in the release notes `http://www.qnx.com/developers/articles/rel_6778_0.html`

3.5 Execution environment

The QOS execution environment includes hardware, firmware, firmware configuration, and when QOS is running as a guest, the underlying hypervisor. System designers are required to perform additional safety analysis as described in the following restrictions:

RST-0001. All users of the QOS SHALL be familiar with the latest version of release notes and bug reports (*QMS3263: Defect Notification*) shipped by BlackBerry QNX for the particular release being used.

If users are uncertain of the implications of a release note on the design or implementation of a system subject to the requirements of IEC 61508, ISO 26262, and IEC 62304, then they should approach BlackBerry QNX for an explanation.

RST-0014. The architecture of a safety application SHALL comply with any and all requirements of underlying components, including the hardware (e.g., processor or SoC) and BSP.

RST-0002. All users of the QOS SHALL be familiar with any errata published for the hardware upon which their application will be run, and SHALL ensure that their application can function correctly in the presence of these errata.

BlackBerry QNX can provide advice and assistance in determining whether a particular hardware platform is suitable for use with the QOS and how individual errata might be mitigated.

RST-0085. System designers SHALL perform a safety analysis of QOS's execution environment and consider the following when performing the analysis:

- hardware errata
- known issues with firmware and firmware configurations, and
- if QOS is running as a guest, known issues with the hypervisor within which it is running

The user SHALL ensure that their application can function correctly in the presence of errata or known issues.

The users must be aware of their hardware platforms and must contact BlackBerry QNX if their safety analysis results in critical issues that might affect the safe operation of QOS.

3.6 Building a QOS Image

3.6.1 QNX Startup overview

The Startup Module prepares the hardware and system page prior to starting the QOS. QOS is designed to be independent of any particular processor and hardware environment. A large part of this abstraction is made possible by the use of a system page (`syspage`), an area of memory initialized by the startup module and used by the QOS to perform hardware-dependent tasks. In addition to basic information about the hardware (amount of RAM, etc.), the `syspage` also contains code snippets known as kernel callouts. These snippets include the code necessary for handling timers, interrupts, power maintenance, cache handling, and debug interfaces and are invoked by the kernel when needed. With the exception of `syspage`, the QOS makes no assumption about the contents of memory when it is first invoked.

The QNX Startup Module is not certified as part of the QOS and could misconfigure the system such that the QOS suffers from latent faults. Due to the complex nature of startup and modern hardware specifications, it is unrealistic to certify the correctness of startup. Therefore, system integrators need to ensure the correct configuration of startup module through system verification. See *QMS3310: Startup Module High-Level Design* and *QMS3435: Kernel Callout Functional Descriptions* documentation for more details.

In a system that uses the QOS, the early system initialization is performed by the startup component of the board support package (BSP). The QOS depends upon the BSP to leave the hardware in an appropriate state, depends upon the BSP to leave a description of the hardware environment in the `syspage`, and depends upon some functionality implemented in the BSP. If these requirements of the QOS are not met by the BSP, the QOS cannot be relied upon to operate as specified.

To illustrate the functional requirements for kernel startup, BlackBerry QNX provides *QMS3310: Startup Module High-Level Design* and *QMS3435: Kernel Callout Functional Descriptions*.

QMS3310: Startup Module High-Level Design:

- Describes the design and the implementation of startup module and the requirements for kernel startup.
- Also, contains a brief description about system page fields that can be modified at runtime.

QMS3435: Kernel Callout Functional Descriptions:

- Documents the requirements of the startup-provided kernel callouts and the execution environment of those callouts.

REC-0004. The recommended command line options in Section A.3.2 SHOULD be used to start `procnto`.

3.6.2 Preparing the system before QOS starts

All applications must be developed according to the following restriction for safe operation of QOS.

RST-0089. System integrators SHALL employ a strategy at the system level to ensure that the program loader and QNX Startup Module have correctly initialized the system.

The startup module sets system internals for the correct operation of the QOS. The startup module is not a certified component. However, BlackBerry QNX provides reference documentation and sample implementations to guide system integrators to correctly configure the startup module.

3.6.3 Kernel Modules

Kernel modules are tightly bound with the kernel and are not normally used for application development. The Adaptive Partitioning Scheduler (APS) is an example of a kernel module.

RST-0140. The QOS configuration SHALL be using the safety variant of the kernel and all (if any) kernel modules.

QOS will drop the 'S' suffix from the output of *confstr()* when called with `_CS_RELEASE`, if any kernel module linked in is not a safety variant. See section 7.3 for more detail about checking the integrity of certified binaries at runtime.

RST-0008. No kernel modules, other than those supplied by BlackBerry QNX, SHALL be added to the QOS.

An inspection of the buildfiles and the output from mkxfs to confirm that only safety-certified kernel modules are linked with the kernel.

RST-0055. The `libmod_qvm` kernel module SHALL be used if the system development also follows the Safety Manual of the QNX Hypervisor for Safety.

3.6.4 Mini Drivers

RST-0052. BSPs SHALL NOT include mini drivers, also known as IDA (Instant Device Activation).

IDA consists of small, highly efficient device drivers that start executing before the OS kernel is initialized. Refer to the link for more details on IDA: `http://www.qnx.com/developers/docs/7.1/#com.qnx.doc.ida.user_guide/topic/about.html`.

If no mini drivers is included, then the system page mdriver size field is 0.

3.6.5 Kernel callouts

RST-0090. System integrators SHALL certify kernel callouts in context with the kernel.

The kernel callouts copied by the startup module are critical for the correct execution of the kernel. A fault in the callouts can cause unpredictable behavior in the kernel. See *QMS3435: Kernel Callout Functional Descriptions* for more details.

RST-0065. The reboot kernel callout provided by the BSP SHALL reset the system in a timely manner.

The reboot kernel callout provided by the BSP is responsible for rebooting or shutting down the system. When the kernel moves to its DSS it cannot force all CPUs to stop executing nor can it guarantee that other CPUs will be prevented from also entering the kernel. QOS also has no control of any pending DMA transfer, so the behavior at the end of these transfers cannot be ensured.

Therefore, the reboot kernel callout is required to reset all processor cores as quickly as it possibly can to return the processor hardware system to a defined state.

In a deployed system, when the reboot callout is invoked by the kernel, the callout must reboot the target regardless of the value of the `abnormal` parameter. The reboot callout parameter `abnormal` is provided only for informational purposes.

A system under development or being debugged may use the `abnormal` parameter to alter the behaviour of the reboot callout. For example, QNX-provided startups have a reboot callout that will reboot the target if abnormal is 0, otherwise, it will spin in an infinite loop; this allows the target to be examined and debugged. QNX-provided startups also support the ability to force a reboot even if abnormal is not 0 [1]. Please refer to *QMS3435: Kernel Callout Functional Descriptions* and system reset section of kernel callouts chapter of SDP user documentation for more details on reboot callout.

3.7 *ClockCycles()* hardware and configuration

The hardware beneath *ClockCycles()* must meet the following requirements:

RST-0015. If QOS is deployed on a multiprocessor system, the system developer SHALL ensure that the hardware used to implement *ClockCycles()* is synchronized across all processor cores.

If *ClockCycles()* is not synchronized across all processor cores, the accuracy of high resolution timers and APS scheduling can be impacted. It has been observed that the TSC ("Time Stamp Counter") is synchronized across all Intel cores within a physical package. However, it has been observed on Intel systems with multiple physical packages that there is an offset between the TSCs of the two packages, and this offset violates Restriction 15.

The *ClockCycles()* function is implemented by reading different hardware clocks on

[1] `SYSTEM_PRIVATE_FLAG_ABNORMAL_REBOOT`, which can be set with the -A parameter

different architectures. In particular, on the `x86_64` architecture, the *ClockCycles()* function depends on the TSC register. As per Intel, the architectural specification for `x86_64` does not explicitly require that the TSC register be synchronized across cores on a multicore processor. However, in practice, for all modern `x86_64` implementations the TSC register is synchronized and this restriction is met.

All ARMv8-A implementations seen by BlackBerry QNX provide per-core clocks that are synchronized across all processor cores.

RST-0083. The *ClockCycles()* frequency SHALL be specified in startup by setting the *qtime* field *cycles_per_sec*.

This can be done either explicitly in code, or, if the startup command line option `-f` is supported to set it, this flag SHALL be specified in the build file.

If no value is specified either hard-coded or using startup command-line options for `qtime->cycles_per_sec`, then startup makes a single attempt to characterize it from other frequency variables, such as the ones for timer and processor, depending upon particular processor and hardware. For example, on Intel, the system tick timer is used as a time reference to measure the period of *ClockCycles()*. Most current startup implementations will, if this value is not provided, attempt to characterize the frequency. However, these attempts to characterize values for frequency variables can be inconsistent across system bootups. Therefore, explicitly specifying the *ClockCycles()* frequency (hard-coding in startup or via a startup command-line parameter) improves QOS's ability to track time more accurately and consistently across system bootups, and may also reduce system boot time.

RST-0084. When porting an old code base to the new QOS release, all uses of the new implementation of *nanospin_ns()* SHALL be examined to confirm that the values given to *nanospin_ns()* continue to meet timing needs.

nanospin_ns() has been changed to spin on *ClockCycles()* instead of inside a for-loop. Due to the weakness of the previous for-loop implementation of *nanospin_ns()*, there may have been user-supplied safety factors, which may lead to either a reduction or extension of the spinning time of the new *ClockCycles()*-based *nanospin_ns()* implementation.

3.8 timecc Kernel Module

`libmod_timecc`, from now on referred to as timecc, is a kernel module that uses *ClockCycles()* as a consistent absolute time source. Instead of using the `timer_load` callout to request an interrupt *N* ns in the future, timecc uses the LAPIC's timer or the ARM PE's Generic Timer to generate system ticks at specific times, based on the absolute value of *ClockCycles()*. See Appendix G for more details on the QOS track of time with and without timecc.

`CLOCK_MONOTONIC` is updated during system ticks using the following mathematical equation:

$$T_{nanoseconds} = (ClockCycles()\ cycles\ -\ boot\ time) * \frac{1\ sec}{qtime->cycles\_per\_sec\ cycles} * \frac{10^9\ ns}{1\ sec}$$

REC-0071. The kernel module timecc SHOULD be used.

timecc improves the QOS's ability to track time, especially when:

- HRTs are frequently used,
- tickless is enabled,
- Über-Authorities interfere with the timely execution of the system tick, and
- synchronization is required between events within QOS and events outside QOS.

RST-0134. If timecc is used in a production QOS system, the safety variant of timecc, i.e., `libmod_timecc-safety.so`, SHALL be used.

3.9 Über-Authorities

3.9.1 Overview

The QOS assumes that it has exclusive ownership and control of the execution environment, which is typically the underlying bare metal hardware platform for Intel x86 processors and ARM-based application (A-series) processors. However, this assumption is invalid for an increasing number of systems. An Über-Authority is hardware and/or software that can deny QOS continuous control over the execution environment, i.e., the cores allocated to QOS. QOS is generally not aware when an Über-Authority seizes control of a processor core or for how long it loses control of the processor core. A system may have any number of Über-Authorities. Examples of Über-Authorities:

- a hypervisor, when running QOS as a guest
- System Management Mode (See *Intel 64 and IA-32 Architectures Developer's Manual: Vol. 3C*)
- TrustZone on ARM platforms
- user code that disables interrupts

The QOS has no control over when and for how long an Über-Authority takes control of a processor core. The Über-Authority can impact all aspects of the system, including interrupt latencies, thread scheduling, APS budgeting, and performance in general. The impacts can propagate to software dependent upon the affected areas of code, and manifest in unknown emergent behavior.

Über-Authorities can introduce new race conditions and deadlocks that otherwise wouldn't occur.

A brief description of Über-Authorities and their potential impacts on QOS can be found in *QMS3528: Spatial and Temporal Isolation*.

A delay in the system tick, for whatever reason (e.g., Über-Authorities, execution in tickless mode, higher priority interrupt compared to the system tick interrupt) may lead to drift in CLOCK_MONOTONIC, CLOCK_REALTIME or cause errors in thread clocks and process clocks, since the system tick is responsible for:

- updating clocks (i.e., CLOCK_MONOTONIC, CLOCK_REALTIME, and thread and process runtime clocks)
- some thread scheduling decisions, and
- firing software timers, including HRTs.

More detail about configuring the clocks and the system tick can be found in the QNX Software Development Platform documentation.

3.9.2 Running QOS on a Hypervisor

The QOS has been tested running on bare-metal hardware on several platforms. The QOS is certified for processor architectures, and the hypervisor presents an architecturally equivalent environment. Therefore, a hypervisor does not invalidate all the safety verification performed on the QOS. In particular, features related to spatial isolation are not expected to be impacted when running on a hypervisor. However, the performance of the QOS and accuracy of time-sensitive features such as APS cannot be guaranteed when running on a hypervisor. The system integrator is responsible for ensuring that the performance of the QOS is sufficiently predictable for their application when running on a hypervisor.

RST-0059. When the QOS is running on top of a hypervisor, users of the QOS SHALL re-verify timing-related safety requirements in the *QOS Hazard and Risk Analysis* that are deemed relevant to the intended safety function.

RST-0060. When the QOS is running on top of a hypervisor, users of the QOS SHALL comply with the safety manual of the hypervisor.

Since a hypervisor is an Über-Authority, see Section 3.9.3 for more guidelines.

3.9.3 General Über-Authorities

An Über-Authority can gain exclusive control over QOS's execution environment. This can impact all aspects of the system, including interrupt latencies, thread scheduling, APS budgeting, and performance in general. See Section 3.9.1 for more details.

RST-0074. To reduce the potential impacts on QOS performance, those Über-Authorities that are not critical to the correct and safe operation of the system SHALL be removed from the system.

RST-0075. When an Über-Authority takes control, it SHALL save the programmer-visible state of all processor cores to be restored before releasing control.

For instance, Intel System Management Mode (SMM), an Über-Authority behaving non-cooperative to the QOS, takes exclusive control of all processor cores upon the triggering of an asynchronous System Management Interrupt (SMI). The QOS requires that the SMM saves and restores the processor state.

3.9.4 Impact on Measuring Time with *ClockCycles()*

ClockCycles() is used as a consistent system-wide time for several purposes, including detecting some activities of Über-Authorities. The following restrictions and recommendations aim to reduce the negative impacts of Über-Authorities on the accuracy of QOS time-keeping.

REC-0073. A system integrator SHALL characterize the drift of `CLOCK_MONOTONIC` against the external environment's time basis over a period of use and under conditions representative of expected use.

If the QOS execution environment's time basis drifts significantly away from that of the external execution environment's time basis, then events initiated within QOS may not be synchronized with the external time basis. The following factors may exacerbate drift:

- high resolution timers
- entering and exiting tickless,
- Über-Authorities, and
- any other system behaviour which may interfere with QOS's time-related activities, i.e., actions that may interfere with the hardware used by QOS for the system tick and *ClockCycles()*.

Note even with timecc there may be a drift. If the frequency of the hardware behind *ClockCycles()* cannot be expressed by an integer, the drift (in cycles) due to this error can be calculated by multiplying the error (in cycles/second) by the maximum running time of the system.

RST-0077. A system designer SHALL estimate the period of time required to program the system tick hardware timer for their hardware.

Note that if the timecc kernel module is used, then Restriction 77 is not applicable.

RST-0076. A system designer SHALL fill in the duration from Restriction 77 into the system page *qtime* field *timer_prog_time* if either:

- the system tick timer is being modified at any time (e.g., HRTs, changes to tickless operation mode, changing *ClockPeriod()*), or
- any Über-Authorities are present.

If the startup supports the `-f` option, it can be used instead of writing the syspage *qtime*. More details can be found in *QMS3310: Startup Module High-Level Design*.

Firing a high resolution timer (HRT), entering and exiting the tickless mode of operation and changing the system tick time (*ClockPeriod()*) all cause writes to the hardware timer used for the system tick.

The execution time can be measured using a high resolution hardware timer, such as the one behind *ClockCycles()*. *ClockCycles()* is implemented as `CNTVCT_EL0` on ARM 64-bit and `RDTSC` on Intel to read the respective processor's timestamp counter.

Instruction and data caches must be flushed prior to each invocation of *timer_load()* in order to measure the worst case execution time. As an alternative to flushing the cache with a flush instruction, the cache may be populated with non *timer_load()* code and data.

Note that if the timecc kernel module is used, then specifying the system page *qtime* field *timer_prog_time* (which can be done with the startup `-f` option) is not required.

RST-0078. A threshold value for the maximum difference between *ClockCycles()* on all cores allocated to QOS SHALL be specified in the presence of Über-Authorities.

The `procnto` option `-c` is used to set a maximum offset threshold value among QOS's cores (see Section A.3.2). At startup, *ClockCycles()* is sampled on each processor core and it has been observed that the *ClockCycles()* is synchronized across all Intel processor cores within a physical package. However, on multiple physical packages, an offset has been observed between the *ClockCycles()* of the two packages, and this offset violates the condition that *ClockCycles()* must be synchronized across all cores (Restriction 15). In addition to *ClockCycles()* offset on multi-cores, non-cooperating Über-Authorities, such as Intel SMM, may suspend one or more cores while QOS is characterizing the *ClockCycles()* offsets. This skew leads to synchronization problems and undefined behavior, such as drifting times and timers.

RST-0079. The sampling with *ClockCycles()* SHALL be based on a hardware-based source.

A hardware-based source for *ClockCycles()* i.e., sampled values using `SYSPAGE_ENTRY(qtime)->cycles_per_sec` not only facilitates detection of Über-Authorities but also improves the ability of QOS to track system time (Restriction 77).

RST-0080. The hardware upon which *ClockCycles()* is based SHALL be synchronized with the time reference of the QOS execution environment, which can be a physical processor or an Über-Authority, such as a hypervisor.

QOS time measurements (and related services such as APS) will be less accurate if the progression of *ClockCycles()* does not match the progression of wall-clock time.

RST-0081. The hardware used to implement *ClockCycles()* SHALL NOT roll over while QOS is in use (i.e., between power up and power off).

The roll-over of hardware for *ClockCycles()* can impact:

- detecting an Über-Authority, and
- determining the correct time

The instance of roll-over can be identified by comparing whether the *ClockCycles()* returned value is less than the previously sampled value. QOS is designed to enter its DSS if it detects that a *ClockCycles()* value is less than the one previously sampled.

If roll-overs were allowed, QOS could not distinguish between one rollover and two rollovers, especially in the case of Über-Authorities.

RST-0082. The frequency of the hardware source for *ClockCycles()* SHALL not vary with the power state of the CPUs and system, for the duration of use from power up to shut down.

Über-Authorities may delay the system tick time significantly, which violates the assumption that a system tick should always be delivered and arrive with minimum delay. This extra delay in the system tick time causes several timing issues, such as a delay in firing timers and scheduling threads. Therefore, a hardware source is required to track the system time accurately because it will provide a reliable time source independent of an Über-Authority.

REC-0060. If an Über-Authority is absolutely necessary, then it SHOULD be implemented as a cooperative Über-Authority, and NOT as a non-cooperative Über-Authority, if possible.

Non-cooperative Über-Authorities can significantly impact the kernel and system performance. For instance, if a non-cooperative Über-Authority gains control over a processor core while that core is executing kernel code, it may preclude all other processor cores from executing kernel code (e.g., making kernel calls)

REC-0045. The current execution state of a processor core (e.g., halted, in privileged state, in user state) SHOULD be taken into account before Über-Authorities attempt to gain exclusive access and control of the QOS.

For example, an Über-Authority could start a seizure immediately after a system tick ISR to reduce the likelihood that the next system tick is delayed.

REC-0046. If the removal of all Über-Authorities is not possible, then the potential impacts of remaining Über-Authorities SHOULD be taken into consideration.

In general, Über-Authorities should exert control for as little time as possible. Similarly, when writing code and choosing compiler optimization options for Über-Authorities, the developer should take into account execution time. For example, Intel System Management Mode (SMM) should take control of the processor cores only as long as it is required.

REC-0047. When setting timeout thresholds for kernel callouts or client-server communication, a designer SHOULD take Über-Authorities into account.

Über-Authorities may reduce or exceed usual timeout thresholds for kernel calls and during a client-server communication. If a kernel timeout is significantly reduced or exceeded from a time specified using *TimerTimeout()*, then it may indicate that control was taken by an Über-Authority.

3.10 System Tick

REC-0074. The system tick interrupt on core 0 SHOULD NOT be delayed for more than half of the tick system interval.

The system tick usually fires periodically but under some circumstances may fire at irregular intervals. The tick interrupt service routine is used by QOS to update time, update thread scheduling information and decisions, fire expired software timers, and prepare and fire high resolution software timers. It plays a vital role in the correct execution of QOS, and any interference with it, or delay of it (e.g., disabling interrupts on core 0), can lead to undesirable behavior.

Tickless operation is supported in QOS and is primarily used to save power consumption by reducing the number of generated system ticks. One valid exception to Recommendation 74 is when tickless mode is enabled in the kernel. See section 4.11.1 for more details.

3.10.1 System tick interrupt

RST-0024. On a multicore processor, the system tick interrupt SHALL be directed to core 0.

The QOS is tested with all interrupts directed to core 0. It is technically possible to direct any interrupt to any processor core. However, the system tick interrupt must be directed to core 0. Other interrupts may be directed to other cores.

RST-0136. A process SHALL NOT attach an interrupt handler to the system tick interrupt, i.e., `SYSPAGE_ENTRY(qtime)->intr`.

CHAPTER 4

QOS Application Development

Contents

This chapter covers aspects of certified application development. It does not cover standard techniques associated with designing and implementing a safe application (for example, avoiding resource deadlocks by creating dependency trees or using techniques such as schedulability analysis) as it is assumed that the reader is familiar with these; only issues specific to the use of the QOS are covered here. BlackBerry QNX provides training that covers many of the more general issues associated with the development of safety-related software.

Section 4.1 programming languages and how to compile an application

Section 4.2 configuration and limitations on the Adaptive Partioning Scheduler (APS)

Section 4.3 issues related to causes of deadlock or livelock including mutexes, condvars, and priority inheritance

Section 4.4 avoiding exhaustion of resources managed by the OS

Section 4.5 issues related to accessing memory without causing interference including DMA access and secure buffers

Section 4.6 issues related to thread management including guard pages and stack allocation

Section 4.7 issues related to interrupts including Interrupt Service Threads, and high-frequency interrupts

Section 4.8 issues related to message passing such as registered events, handling request timeouts, and pulse pools

Section 4.9 signal handling in certified applications

Section 4.10 event tracing in certified systems

Section 4.11 power management in certified systems including hardware features and tickless mode

Section 4.12 floating point emulation in certified applications

Section 4.13 custom kernel call in certified applications

4.1 Writing and compiling applications

4.1.1 Programming languages

REC-0031. A developer SHOULD NOT use inline assembly in the development of an application for a QOS system.

The use of inline assembly is discouraged, as the practice is highly error prone. Errors introduced in inline assembly are often subtle and difficult to detect.

4.1.2 Compiling an application

RST-0122. Only QNX qualified tools SHALL be used to modify compiled binaries.

The following QNX utilities modify the binary and are qualified to TCL3 and T3:

- `C` and `C++` compilers (`qcc/q++`), linker and assembler.
- `ar`, a utility to create, modify and extract archives.
- `objcopy`, a utility to copy the content of one object file to another
- `strip`, a utility to remove symbols and relocation information from executables and shared objects
- `mkifs`
- `secpolcompile`

Refer section: 5.8 for more details.

RST-0007. All developers creating applications to run on the QOS SHALL be aware of the requirements and recommendations concerning command-line options, provided to components and utilities of the QOS, contained in appendix A on page 87. Developers SHALL follow the requirements contained in that appendix and SHOULD follow the recommendations contained in that appendix.

RST-0063. The `qcc/q++` compiler SHALL be used to compile safety critical applications.

RST-0110. All safety critical applications SHALL be compiled with the `qcc/q++` compiler using only the options listed in A.1.1.

RST-0111. No compiler builtins SHALL be explicitly called from any application that is compiled with the `qcc/q++` compiler.

Generally, the compiler builtins names are prefixed with double underscores and can be identified by their names. Note the `qcc/q++` compiler may utilize some compiler builtins when building the QOS with allowed compiler options. For example, the fortified system function feature can be enabled by specifying `-D_FORTIFY_SOURCE=2` preprocessor option, which requires compiler builtins. Those compiler options are in the safety verification scope of the QOS, but are not allowed to be explicitly called by any application. However, the `libm` library is safety-certified with compiler builtins disabled and therefore the `libm` library is recommended to be compiled in a separate compilation unit than the QOS. See *QMS33359: libm Safety Manual* for more details.

RST-0086. Any shared or static library linked or loaded by a certified application running on QOS SHALL be certified to the same SIL/ASIL level as the application.

RST-0121. All projects using QOS 2.2.2 SHALL compile or recompile all binaries and their composite object files using the latest version of QOS 2.2.2 toolchain.

4.1.3 qcc/q++ alignment specifier warnings

`qcc/q++` does not, in all cases, generate a diagnostic warning for code where the defining declaration of an entity has an alignment-specifier, and a non-defining declaration of that entity specifies a non-equivalent alignment. Other methods of detection must be used. Document *ISO/IEC 14882:2011*, 7.6.2 specifies an Alignment-specifier.

4.2 Adaptive Partition Scheduling

The Adaptive Partition Scheduling (APS) allows threads to be allocated to scheduling partitions (See the Adaptive Partitioning User's Guide for an overview `http://www.qnx.com/developers/docs/7.1/#com.qnx.doc.adaptivepartitioning.userguide/topic/ap_overview_Base_.html`)

The following guidelines enable users to configure Adaptive Partitioning Scheduling APS in a safe manner.

RST-0124. A maximum of 8 APS partitions SHALL be configured on QOS.

RST-0026. When using APS, if the design of a safety feature requires accuracy in the minimum partition budget for correct operation, then the safety feature SHALL be tested under high load conditions.

The accuracy of the minimum budget size within APS is dependent upon many variables. The intent of the APS design is to achieve a guarantee that the actual minimum budget of a partition is within 1 percentage point of the configured minimum budget (that is, a partition configured with an X% minimum budget shall receive at least X-1% of the processor resources available within a scheduling window, presuming that processes running within the partition can make use of the budget) under the following conditions:

- the configured minimum budget is >= 5%
- the system has 8 or fewer processor cores
- critical budgets are not used
- clock tick interrupts occur at a regular interval
- the window size is configured to ensure at least 200 scheduling events (thread-blocking events or timer ticks) per processor core per window

If these conditions are not met, APS will continue to function but the partitioning of the processor core according to minimum budget specification will be less accurate.

To ensure that your critical applications receive the processor cycles that they require, you should verify the correct operation of your applications under high processor load.

RST-0027. The application SHALL NOT rely on the correct detection of bankruptcy by the Adaptive Partitioning Scheduling (APS).

RST-0028. The APS scheduler SHALL NOT be used on multicore systems that provide different processing power on different cores (that is, asymmetric multicore processors).

RST-0029. A system SHALL NOT combine the use of the APS scheduling algorithm and power management where different cores can be configured to have different power levels.

RST-0100. When APS is used, APS security SHALL be enabled using the *SchedCtl()* command SCHED_APS_ADD_SECURITY, and SHALL be enabled immediately using the highest security setting appropriate for the system.

APS security is disabled by default, therefore any process can interfere with other processes by creating a new partition, or modifying or joining existing partitions.

In the absence of the APS scheduler, the QOS implements strict priority scheduling – that is, the highest priority thread that is ready to run will run. However, the scheduling partitions enforced by APS have precedence over thread priority. In a system running APS, if the highest priority thread that is ready to run belongs in a partition that has no scheduling budget left, it will not be selected to run if another partition needs processor time.

The precision with which the APS operates depends on the resolution of *ClockCycles()*.

REC-0013. The APS scheduler can be configured dynamically, but this is not recommended. The APS configuration SHOULD be established at system startup. Once a system has begun providing a safety function, APS SHOULD NOT be reconfigured.

REC-0014. The system designer SHOULD avoid the use of APS critical budgets.

REC-0054. Immediately after APS has been configured, it SHOULD be locked using SCHED_APS_SEC_PARTITIONS_LOCKED to prevent any further unauthorized changes.

4.3 Deadlocks, livelocks, and priority inversion

4.3.1 Priority inheritance

REC-0020. Threads that are subjected to different priority limits SHOULD NOT share resources that provide priority inheritance (e.g., mutexes).

An unprivileged process cannot have threads running above the maximum priority set for such threads (default is 63). If a resource is locked by an unprivileged thread with a low priority when a privileged thread running at priority higher than the unprivileged priority limit tries to lock that resource, then the QOS's priority inheritance mechanism will raise the unprivileged thread's priority to the maximum unprivileged value. This results in a priority inversion: the thread holding the mutex has had its priority boosted to the maximum unprivileged priority, but this priority is not as high as the priority of the thread waiting on the mutex.

Also note that:

- processes can set their threads to a privileged priority only if they have the `PROCMGR_AID_PRIORITY` ability.
- As specified in the user documentation, using the `_NTO_CHF_FIXED_PRIORITY` flag on a call to *ChannelCreate()* disables priority inheritance on that channel: the server does not acquire the priority of the client. This makes the server susceptible to priority inversion.
- Internally, the `pipe` mechanism relies on semaphores and other synchronization objects for which priority inheritance is not provided. Therefore, a design where an application can be locked when writing to a pipe cannot rely on priority inheritance occurring.

4.3.2 Mutexes

REC-0021. The QOS SHOULD be configured to ensure mutexes that are shared between processes are safe.

Note that in order to share a mutex, two processes must also share the memory in which the mutex resides.

If an application needs to share a mutex between separate processes, then it must either expose itself to potential interference from unrelated processes, disable priority inheritance, or accept a performance penalty as all operations on the shared mutex must enter the kernel.

Normal kernel configuration does not require that all mutex operations enter the kernel. This enhances the performance of mutex operations but, in such a system, a thread that corrupts a mutex (accidentally or maliciously) can cause the priority of an unrelated thread to be boosted to the first thread's priority through the priority inheritance mechanism.

The kernel can be configured to reject attempts to lock shared mutexes that would cause a priority inheritance unless all mutex locking operations enter the kernel. This has a significant impact on the performance of shared mutexes, but guarantees that non-cooperating threads cannot interfere with each other. In order to use shared mutexes in a safe manner:

- The system must be configured to require that all mutex operations on shared mutexes that support priority inheritance must enter the kernel. This may be achieved by specifying the `-s` command-line option to `procnto`. (Note that this is the default for the QOS; see section A.3.2 `procnto` command-line options recommendations.)
- All mutexes that are to be shared between processes must either disable priority inheritance by setting the `PTHREAD_PRIO_NONE` flag or must be explicitly identified as a shared mutex by setting the `PTHREAD_PROCESS_SHARED` flag.

In a system where the `-s` option is specified, an attempt to lock a shared priority-inheriting mutex that does not have the `PTHREAD_PROCESS_SHARED` flag set will fail and return an error code.

4.3.3 Mutex and condvar handling

- In accordance with the POSIX standard, if a thread dies while holding a mutex, then that mutex will never be released. A mutex recovery protocol does exist: see the library functions *SyncMutexEvent()* and *SyncMutexEvent_r()*.
- If a condvar is destroyed by one thread while another thread is waiting for it, POSIX does not define the action to be taken. The QOS destroys the condvar and leaves the waiting thread indefinitely blocked.
- An application should not unmap a memory page that contains a locked mutex. The QOS handles this situation correctly—the ownership of the mutex is removed from the locking thread, threads waiting on the mutex with a timeout are released with

an error code, and threads waiting on the mutex without a timeout are terminated. However, this behavior has been found in the past to be confusing for application designers.

4.3.4 Resource manager framework

REC-0023. Any resource manager invoked from another resource manager SHOULD be engineered in such a way that it always has at least one thread RECEIVE-blocked.

This can help prevent deadlock scenarios.

4.4 System resource Management

RST-0011. The system SHALL be engineered in such a way that exhaustion of any resource, in particular memory, does not occur.

Memory exhaustion generally does not occur with safety-related systems where memory is statically allocated at application startup and lazy allocation of memory (e.g., for stacks) is disabled—see Section A.3 for details on disabling lazy stack allocation.

The QOS provides a number of constraint mechanisms that prevent applications from accessing or consuming more than a configured amount of different resources. These constraint mechanisms include rlimits (see the POSIX *setrlimit()* function), procmgr abilities (see the QNX *procmgr_ability()* function), POSIX permissions, and APS.

The resources, the constraint mechanisms, and the type of the resource (transient versus non-transient, i.e., resources that are cleaned up automatically on the death of the process that allocated the resource, versus resources that survive the death of the process that allocated them) are given in Appendix H.

REC-0006. Applications SHOULD allocate resources required to implement safety functions early in their life cycle. In particular, large memory allocations, allocations of physically contiguous memory and any other resource on which the operating system does not impose constraints SHOULD be completed at application startup.

Refer to Appendix H for further detail.

4.5 Memory access

4.5.1 Mapping to Fixed Addresses

QOS allows processes to create mappings to specific virtual addresses, and it allows privileged processes to map to virtual address 0. The application developer should consider the following recommendation when processes have abilities to map fixed memory address ranges.

REC-0050. Processes mapping memory with `MAP_FIXED` and having the `PROCMGR_AID_MAP_FIXED` ability SHOULD NOT create mappings at virtual address 0.

Note that a memory mapping at virtual address 0 is only possible if a call to *mmap()* is made with `MAP_FIXED` while having the `PROCMGR_AID_MAP_FIXED` ability. If a mapping is created to virtual address 0, this breaks any code that assumes a SIGSEGV will be generated when dereferencing a NULL pointer.

4.5.2 DMA access

RST-0021. A system that includes hardware apart from the processor that has direct access to the system's memory (such as a bus master DMA device) SHALL employ hardware-level memory access constraints (such as an IOMMU) to ensure sufficient isolation between the separate applications and between the applications and the operating system.

See also section *5.5 SMMUMAN*.

However, if there is no IOMMU on the system but an equivalent hardware and software to make use of protected DMA accesses, the system designer should make sure it is certified to the highest SIL/ASIL level.

4.5.3 Secure buffers

Shared memory objects can be either named or anonymous. Named shared memory objects can be opened using known public paths (/**dev**/**shm**) by calling the POSIX *shm_open()* function. Since they are visible to all processes, it is possible for a process with the correct file permissions, but not involved in a specific client-server relationship, to access the shared memory object and modify it, thereby interfering with the processes sharing the object. Anonymous shared memory objects can be shared without a public pathname and can be typically shared only among processes with a common lineage, i.e., parent-child relationship.

A secure buffer is a memory object that the creating process may share with both child and non-child processes.

4.5.3.1 Secure buffer control flow example

A typical way of working with a secure buffer is:

- The client requests a buffer from the server
- The server (producer) creates an anonymous shared memory object (secure buffer) by calling *shm_open()* with `SHM_ANON` flag, receiving a file descriptor (`fd`) from QOS
- The server creates a handle, by calling *shm_create_handle()*, for the secure buffer for the client (consumer) using the *fd* and the client's pid
- The server may impose other restrictions on the use of the handle or shared memory object by the client, such as specifying it's a revocable object with the `SHMCTL_REVOCABLE`
- The server replies to the client buffer request with the handle

- The client uses the handle to access the secure buffer (i.e., *mmap_handle()*, or *shm_open_handle()*
- When finished, the client notifies the server that it's done with the secure buffer
- The server revokes the client's access to the secure buffer using *shm_revoke()*

After the client has a file descriptor *fd*, the server cannot prevent the client from mapping the *fd* (with *mmap()*), and also accessing the revoked secure buffer. A client receives a SIGBUS when attempting to access a revoked secure buffer.

4.5.3.2 Secure buffer limitations

A secure buffer and its handle can be created write-only. However, on all supported hardware platforms, due to the limitations of the processor's MMU, the mapping returned is not only writable but also readable. See the details of the following functions in the QNX Software Development Platform documentation:

- *shm_create_handle()*
- *mmap()*
- *mmap_handle()*

QOS is capable of creating up to 2^{64} secure buffer handles. The QOS will enter its DSS if it runs out of secure buffer handles.

Secure buffers provide safe anonymous shared memory objects that can be shared between any two processes. A system configuration error could cause an unsafe memory access to a shared buffer. The following recommendations reduce the likelihood of misconfiguring the shared memory objects.

REC-0048. If any process maps to specific physical address ranges, then those ranges SHOULD be checked for unintended overlap with address ranges assigned to other processes.

REC-0049. Typed memories (created by calling *posix_typed_mem_open()*) SHOULD be used instead of hard-coded physical addresses, where possible.

A shared memory buffer created by mapping a specific physical address can be dangerous. QOS does not track the physical addresses mapped using (*mmap()* or *mmap_handle()*). It is possible for two or more processes with the ability to map specific physical addresses (`PROCMGR_AID_MEM_PHYS`) to interfere with each other. An example of a possible scenario to avoid is:

1) Client C1 requests a secure buffer from server S1
2) S1 creates a secure buffer on physical memory at physical address PA
3) S1 shares the secure buffer with C1 by creating a handle for C1
4) S1 sends the handle to C1
5) C1 consumes the handle by mapping it (*mmap()*)

6) S1 terminates abruptly without revoking any secure buffers
7) The server is restored by the system. Let's call this new server process S2
8) Another client C2 requests a secure buffer from S2 and it creates a secure buffer on the same physical address PA
9) S2 shares the secure buffer with C1 by creating a handle for C2
10) S2 sends the handle to C2
11) C2 consumes the secure buffer handle by mapping it

Now both C1 and C2 have mappings to the same physical memory and are likely to interfere with each other.

4.5.4 *mmap_peer()*

mmap_peer() allows any process with the `PROCMGR_AID_MEM_PEER` ability to create mappings in the virtual address space of another process. The use of *mmap_peer()* is generally discouraged, and secure buffers should be used instead. However, some system integrators may choose to use *mmap_peer()* for performance reasons under some circumstances.

Certain restrictions apply to the use of the *mmap_peer()* and *munmap_peer()* functions to ensure freedom from interference in a mixed-criticality environment.

RST-0068. A process SHALL NOT *mmap_peer()* memory into the address space of another process with a higher SIL/ASIL.

RST-0069. A process SHALL NOT *mmap_peer()* into the address space of another process with a lower SIL/ASIL if the corruption of that memory could lead to incorrect behavior of another process with a higher SIL/ASIL.

RST-0093. A process using *mmap_peer()* SHALL have its `PROCMGR_AID_MEM_PEER` ability restricted (i.e., specifying a subrange of peer processes for the ability) such that it may not *mmap_peer()* into a process with a higher SIL/ASIL.

4.5.5 Memory exhaustion and floating point exceptions

The QOS kernel will enter its DSS if it receives a SIGFPE due to memory exhaustion while attempting to allocate an FPU context structure for a thread. By default, the QOS kernel will allocate an FPU context when creating a thread. When the `-~fa` option is passed to `procnto` then the context will be created when the thread first attempts to use the FPU.

4.5.6 Cacheable and uncacheable memory

Memory may be mapped as cacheable or uncacheable. Changing memory between cacheable and uncacheable in an executing system should be handled with care to preserve the consistency of the memory between the cacheable view and actual memory, and between the views from different processing cores.

Use of the *mprotect()* API to change the cacheable attributes of memory should be avoided.

4.6 Thread management

4.6.1 Guard Pages

QOS automatically creates a stack for a user thread and places a “guard page” at the bottom of the thread’s stack to detect when the thread has exceeded the limits of its stack. It is also possible for a user to provide a region of contiguous memory to be used as a thread’s stack. The following restrictions guide users to correctly configure the size of guard pages for a manually created thread stack.

RST-0098. If the thread’s stack is specified using *pthread_attr_setstack()*, *pthread_attr_setstackaddr()*, or *ThreadCreate()*, then the thread stack SHALL have a guard page of at least `_SC_PAGE_SIZE` bytes immediately before the lowest stack address.

Typically, the stack size is specified as a multiple of `_SC_PAGESIZE` and the TOP and BOTTOM addresses are page-aligned (i.e. ((TOP + 1) mod `_SC_PAGESIZE`) and (BOTTOM mod `_SC_PAGESIZE`) are 0). The guard page has a finite size, and this will very likely detect accidental stack overflow, however it is theoretically possible to skip the guard page if a function:

- creates a stack frame that spans the guard page (or extends its frame to span the guard page (e.g. *alloca()*), and
- the guard page is not accessed during the life of the function, and
- the region beyond the guard page is mapped such that accessing it will not fault.

This scenario will allow the function to corrupt the region beyond the guard page.

RST-0099. The guard page size for a QOS-provided thread stack SHALL NOT be set to 0, i.e., the guard page size SHALL be either left at the default size, or set to a value provided that is greater than the default size.

The guard page size can be set using *pthread_attr_setguardsize()*.

4.6.2 Manual thread stack allocation

REC-0026. Applications SHOULD free stack memory when manually allocating thread stacks.

If the system allocates a stack, it reclaims the space when the thread terminates, whereas manually allocated stacks must be freed manually as well. See *Getting Started with QNX Neutrino* for details on *pthread_attr_setstackaddr()* and manual stack allocation: `http://www.qnx.com/developers/docs/7.1/#com.qnx.doc.neutrino.getting_started/topic/s1_procs_Thread_attributes_stack.html`.

4.6.3 Thread cancellation

REC-0032. A developer SHOULD NOT use thread cancellation in the development of a safe application.

The use of thread cancellation is highly discouraged, as the practice is highly error prone. It can lead to unintended resource leaks and unknown execution behaviors. Errors introduced with thread cancellation are often subtle and difficult to detect and thus we recommend using a controlled and architected thread termination lifecycle.

4.6.4 Thread Scheduling on multicore System

4.6.4.1 Higher-priority threads are not always running

This section describes some scenarios when higher-priority threads are ready to run while lower-priority threads are running.

The *N* highest priority threads on an *N* core processor are not always running in an SMP environment. In some situations, a lower-priority thread is running when a higher-priority thread is ready to run but not running. For instance, consider the following scenario:

- a thread T_H is currently running on core *A* at priority *H*
- a thread T_M has been defined such that it may only be run on core *A* in an SMP environment and has priority *M* where $M < H$
- a thread T_L is running on core *B* at priority $L < M$
- T_M was waiting for an event that has now occurred and is now ready to run
- T_M can only run on core *A* and since core *A* is occupied with a higher priority thread, T_M will not execute until T_H reduces its priority or blocks
- although $M > L$, thread T_L is not preempted because T_M cannot run on core *B*.

In principle, T_H could be moved to core *B*, preempting T_L, making core *A* available to T_M. Even if core *B* is idle, T_H will continue to run on core *A*, and T_H will migrate only if preempted by a higher priority thread. Similarly, a client thread unblocked from *MsgReply()* can remain ready to run until the server calls *MsgReceive()*, while lower priority threads (even the idle thread) are running on other processor cores.

4.6.4.2 Scheduling latency

It is quite possible for a thread to be in READY state or preempt a lower-priority thread even when there are idle cores available. A thread may stay in a READY state before it is migrated to an idle core. Following are some possible scenarios when this behavior can occur:

- The thread scheduler simultaneously moves two threads on separate cores to the head of each core's ready queue. There are two other cores that are idle. The thread scheduler on each of the cores with the ready threads, simultaneously picks the same idle core to signal with an IPI. This idle core receives the requests and handles them in the order it receives them. As such one of the ready threads is transferred to the signalled idle core where it runs, but the other ready thread remains in the ready state despite there being another idle core available.

- Similarly, two threads are made ready by the thread scheduler one after another on the same core, and both times an IPI is sent to the same idle core. Only one of the two ready threads runs, while there are other idle cores.
- Thread T_A with priority *A* is made ready by the thread scheduler on some core that sends an IPI to core 0, which is idle. Before the IPI is handled, core 0 takes an interrupt that readies thread T_B with priority *B* where $B >= A$. T_B is made running on core 0. When the kernel takes the rescheduling decision, T_B continues running and T_A remains ready.
- Thread T_A with priority *A* is made ready on some core by the thread scheduler that sends an IPI to core 0, which is idle. Before the IPI is handled, core 0 takes an interrupt that readies thread T_B with priority *B* where $B < A$. T_B is made running on core 0, since core 0 is idle. When the kernel takes the rescheduling decision, T_A is made running and T_B becomes ready.

4.7 Interrupts

4.7.1 Interrupt Service Threads

REC-0062. *InterruptAttachEvent()* SHOULD be used instead of Interrupt Service Routines (ISRs).

InterruptAttach() allows an ISR to be registered; i.e. it causes the provided function to be called by the QOS interrupt-handling infrastructure while the processor core is in a privileged context. Any errors in the provided function will likely have a significant impact on the correct execution of QOS and all other processes.

InterruptAttachEvent() causes a `sigevent` to be delivered to unblock a user thread when an interrupt arrives (known as an Interrupt Service Thread (IST)). The IST, like any thread, remains isolated from other processes and the kernel.

Some reasons ISTs should be considered over ISRs:

- ISTs run in user space while ISRs run in kernel space. ISRs make use of the kernel stack and can potentially cause stack overflow. Using ISTs maintains spatial isolation between user code and the kernel.
- ISTs have a shorter servicing path, i.e., there is only one context switch from the currently running thread to the IST, whereas with an ISR, the ISR context must be set up, executed, kernel re-entered, and then follow up activities in user space scheduled.

A system with low-frequency interrupts should consider ISTs. A system with high-frequency interrupts should consider frequency of interrupts, latency, and hardware buffering before choosing ISTs or ISRs.

4.7.2 High-frequency interrupts

RST-0017. The application SHALL NOT contain a continuously running, regular, high-frequency hardware interrupt unless the `-p` command-line option is used when starting the kernel.

The specific minimum frequency that can be a problem depends upon the processor on which the application is running and upon the quantity of different kernel resources that are consumed. However, an interrupt with a frequency similar to the timer tick (typically 1 kHz) is known to be safe on all modern hardware in all circumstances. If there is any concern about the frequency of an interrupt, the system should be verified for correct behavior under maximum load.

Also, see section *8.6 Kernel call latency, duration, and pre-emption*.

4.7.3 Non-Maskable Interrupts

REC-0063. Non-Maskable Interrupts (NMIs) SHOULD NOT be used.

Eponymously, NMIs cannot be masked; the QOS supports NMIs but treats them as it does other interrupts—they are not given any special priority over other interrupts. This may be unexpected in a system design.

4.7.4 `SIGEV_THREAD` Sigevents

RST-0101. Sigevents of type `SIGEV_THREAD` SHALL NOT be used.

In low memory situations, a `sigevent` of type `SIGEV_THREAD` being delivered in an interrupt context (i.e., timer firing, or interrupt service routine returning a `sigevent`) may fail to create a thread, and this currently results in a silent failure.

4.8 Message Passing

4.8.1 Registered events

A signal event, `sigevent`, or `event`, enables QOS to pass information from one process to another (or itself) in the form of a notification. QOS requires that events be associated with either a specific connection or all that process's connections. Such events are known as registered events. Registered events prohibit servers sending unnecessary events to clients, and therefore, avoid undefined behaviors. For example, on a connection that accepts unregistered events, a process with a channel ('server') can deliver any kind of event to any process that has a connection to that channel ('client'). If the client is not configured to handle the unregistered events, this may lead to undefined behavior. For instance, a client may receive a message from the server that was originally intended for a different client.

By default, all events sent over a connection must be registered events. However, there may be a legitimate need to allow unregistered events for a specific connection.

A connection can be configured to accept unregistered events by using the flag _NTO_COF_UNREG_EVENTS. However, the use of this flag is discouraged in production.

RST-0107. A client SHALL NOT register an event for all servers (i.e., pass a *coid* of -1 to *MsgRegisterEvent()*) unless the client is prepared to receive that event from all processes to which it establishes a connection.

If a client registers an event for all servers then the client may receive that event from any server it is connected to, including servers from which it is not expecting that event.

REC-0043. A developer SHOULD NOT create a client connection with a server enabling the delivery of unregistered events. A client can allow the delivery of unregistered events by setting the flag _NTO_COF_UNREG_EVENTS when it calls *ConnectAttach()* to connect to server.

4.8.2 Client timeouts

Many kernel operations—e.g., named semaphores, POSIX message queues (implemented with `mqueue` and `mq`)—use the kernel's messaging system "under the covers". This means that, if a client-defined timeout occurs before the server has had the opportunity to handle the request, the server will never see the request: the message pass will be stopped and the client released.

4.8.3 Setting robust timeouts for blocking function calls

A blocking function call may internally send several messages to one or more servers. The *TimerTimeout()* function cannot be used to set a timeout for such a function because the timeout will only apply to the first message sent.

In such cases, a timer should be registered with an event to unblock the thread prior to calling the function. After the function returns, the timer is disabled.

For example:

```
static timer_t const id = -1;

int start_timeout(uint64_t const timeout) {
    struct sigevent ev;
    SIGEV_UNBLOCK_INIT(&ev);
    if (id < 0) {
        id = TimerCreate_r(CLOCK_MONOTONIC, &ev);
    }
    assert(id >= 0);
    struct _itimer max_time;
    max_time.nsec = timeout;
    max_time.interval_nsec = 1000000;
    return TimerSettime_r(id, 0, &max_time, NULL);
}
```

```
void stop_timeout() {
    struct _itimer max_time;
    max_time.nsec = 0;
    max_time.interval_nsec = 0;
    TimerSettime_r(id, 0, &max_time, NULL);
}
```

A more complex implementation would be required for multi-threaded programs to handle multiple timer contexts.

4.8.4 Pulse Pool

When a pulse is delivered to a channel, if there is at that time no thread receive-blocked on that channel, the kernel must enqueue the pulse for later delivery. Doing this requires an allocation of a small amount of memory, which, by default, the kernel will take from system memory. In low memory conditions, if the allocation fails, the kernel will enter in to its DSS. However, when a channel is created with a pulse pool, and if a pulse must be enqueued, then the kernel will allocate only from the channel's pulse pool. If a pulse pool is depleted, then the kernel will terminate the process that created the channel, but the kernel will not enter into its DSS.

REC-0067. Any channel created by a process SHOULD be created with a pulse pool of size dependent upon:

- the consumption rate of service thread(s) receiving on the channel,
- the number of clients, and
- the characteristics of communication between client and server.

RST-0125. If a channel is created with a pulse pool, and the flag `_NTO_CHO_CUSTOM_EVENT` is passed to request notification of failure to deliver a pulse, then:

- a sigevent of type `SIGEV_SEM` SHALL be used, and
- the rearm threshold SHALL be set to a value greater than the number of pulses in the pulse pool.

A sigevent of type `SIGEV_NONE` allows a channel to silently drop pulses, but dropping pulses can leave the system in an inconsistent state. For example:

- Dropping a pulse from an interrupt sigevent could cause a driver to leave an interrupt masked.
- Dropping a disconnect pulse could leak a server connection ID (scoid) or client entry in a server (e.g., a resource manager).

RST-0141. Unless a design explicitly tolerates dropped pulses, a designer SHALL ensure that if a notification of a dropped pulse is received, then:

- a notification of a dropped pulse SHALL cause the channel's process to enter its DSS, and

- that a sigevent of type `SIGEV_SEM` SHALL be used in lieu of `SIGKILL` only to perform a small amount of cleanup before entering the DSS.

Receiving notification of a dropped pulse is an indication of a fundamental design problem, for instance:

- the process is not consuming quickly enough,
- the producers are producing too quickly,
- there are too many producers, or
- the pulse pool is too small.

4.9 Signal handling

Signals asynchronously interrupt a process (or specific threads within a process as described below) which can lead to various problems:

1. Inconsistent data state: A signal handler could operate on data that the thread was already manipulating. Since the signal delivery basically adds another execution thread to the receiver the potential data state inconsistency can't be solved with mutexes.
2. Unexpected failure/retry of operations.

REC-0064. A system SHOULD use POSIX signals only either:

- synchronously on the receiver end, i.e. masked at all times and only received by *sigwaitinfo()* or equivalent, or:
- when used to terminate processes.

In general, when a signal is delivered to a process, the decision as to which thread the incoming signal is delivered is made as follows:

1. If the signal is thread-specific (e.g., SIGSEGV), then it is delivered to the appropriate thread.
2. If a thread is waiting on a call to *SignalWaitInfo()* or *SignalWaitInfo_r()* that has specified the particular type of signal, then the signal is delivered to such a thread.
3. Otherwise, the signal is delivered to an arbitrary thread that is not blocking it. This thread may not be the highest-priority thread, and so the signal may remain queued while a higher-priority thread continues to run. This situation does not occur for SIGKILL, as that signal is delivered to all threads.

If a thread calling *waitfor()* receives a signal, it returns immediately with a return code of -1 — i.e., the *waitfor()* is interrupted.

4.10 Event tracing

Tracing events with kernel instrumentation should be used only for debugging purposes. Tracing has an unpredictable and severe impact on system performance. The ability to enable tracing should be restricted outside of a debugging context.

RST-0072. A system SHALL NOT have any processes with the `PROCMGR_AID_TRACE` ability in normal mode of operation.

4.11 Power management

4.11.1 Tickless

The kernel can reduce power consumption by running in tickless mode. When tickless is enabled, QOS will—when the system is idle—automatically reduce the number of system ticks generated.

In order for the kernel to enter tickless mode, the following must all be true:

- Tickless operation must be enabled by setting the `QTIME_FLAG_TICKLESS` flag in the `qtime` member of the system page
- The clock `CLOCK_REALTIME` must not be in the process of being adjusted because of a call to *ClockAdjust()*
- All processor cores must be idle

After the last processor core goes idle, the hardware timer used for system ticks is programmed to generate an interrupt just after the next software timer is set to fire. Normal ticking is resumed when a non-idle thread becomes ready. This will be caused by an interrupt (either the system tick or a peripheral) causing a thread to become ready to run.

When QOS resumes normal ticking, or fires a high resolution timer (HRT), the clock phase may shift from the previous normal ticking phase. The phase shift may cause a timer to fire up to one clock period earlier or later than it otherwise would have before entering into the tickless mode. This timer delay may have undesirable side effects for application timing. See Figure 4.1 for details.

When QOS enters and exits tickless mode, some overhead is involved which may cause delays to:

- interrupt service routines (ISR),
- firing of software timer sigevents, and
- resumption of threads:
  - those made ready by software timers firing, and
  - interrupt service threads (ISTs).

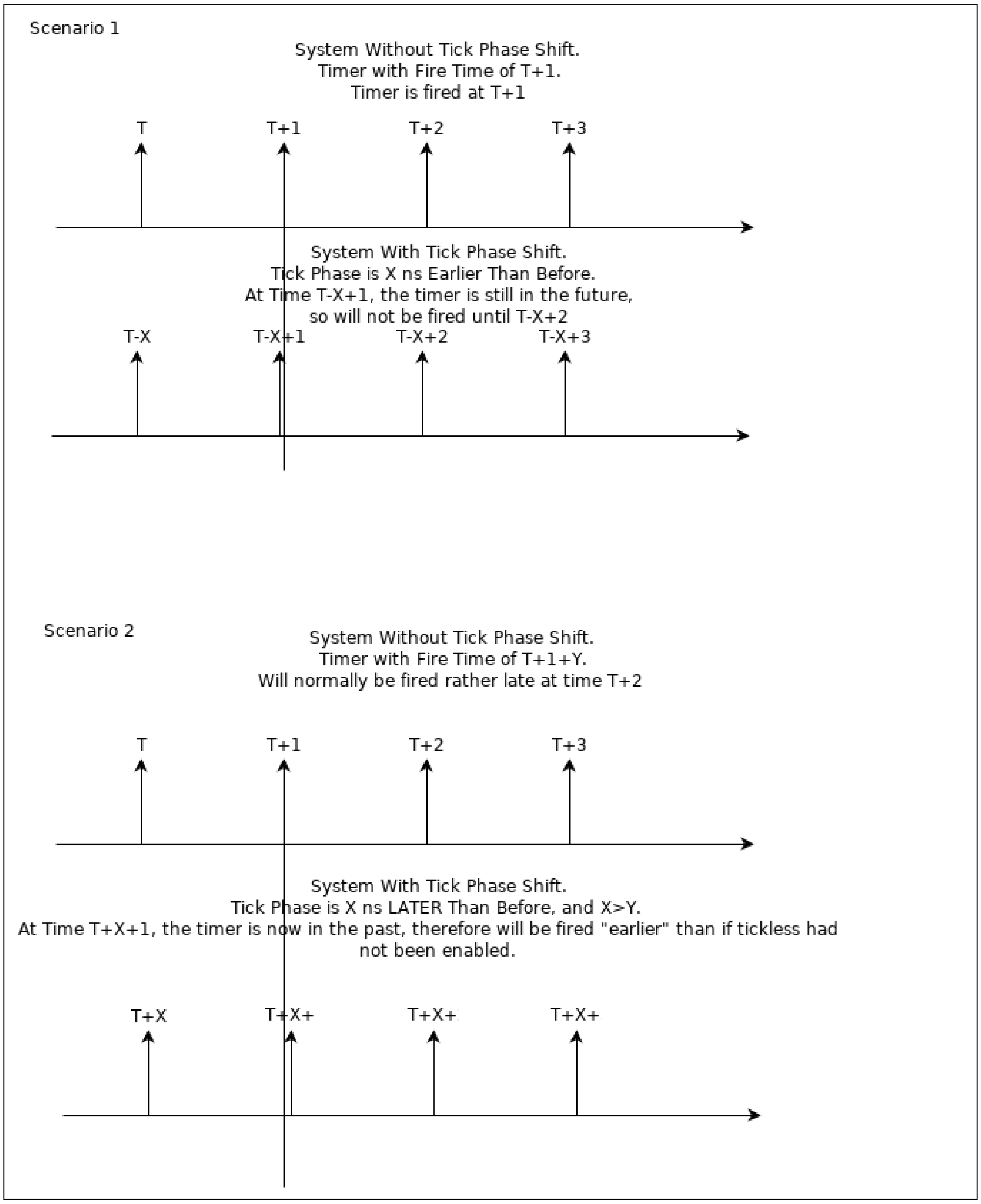
Scenario 1
System Without Tick Phase Shift.
Timer with Fire Time of T+1.
Timer is fired at T+1
T
T+1
T+2
T+3
System With Tick Phase Shift.
Tick Phase is X ns Earlier Than Before.
At Time T-X+1, the timer is still in the future,
so will not be fired until T-X+2
T-X
T-X+1
T-X+2
T-X+3
Scenario 2
System Without Tick Phase Shift.
Timer with Fire Time of T+1+Y.
Will normally be fired rather late at time T+2
T
T+1
T+2
T+3
System With Tick Phase Shift.
Tick Phase is X ns LATER Than Before, and X>Y.
At Time T+X+1, the timer is now in the past, therefore will be fired "earlier" than if tickless had
not been enabled.
T+X
T+X+
T+X+
T+X+

Figure 4.1: Tickless Execution Timing Scenarios

In tickless mode, QOS reduces the number of ticks when the system is idle in order to reduce power. Depending on the timing and tolerance of software timers, or when a waking interrupt arrives, tickless mode may affect normal ticking operations by shifting the phase of the system tick interrupt. This can affect the firing of timer events, causing a timer to fire anywhere between its specified fire time, and up to one clock period later than if the phase shift had not occurred. When compared to a system with tickless disabled, the overhead of exiting tickless may cause an increase of the latency of a waking interrupt and any thread it causes to be scheduled.

RST-0092. In tickless mode, an interrupt service routine (ISR) SHALL NOT make use of *SYSPAGE_ENTRY(qtime)->nsec*.

While tickless, if an ISR reads the system time *SYSPAGE_ENTRY(qtime)->nsec* then it may sample a stale system time because time is not updated until normal ticking is resumed.

4.11.2 Dynamic power management

REC-0008. A system SHOULD NOT make use of dynamic power management.

That is, the dynamic changing of processor core speeds or the dynamic turning off of processor cores. This type of operation is unusual for safety-critical systems, but fairly common in other applications to extend the life of the processor or reduce power consumption. Its use is not prohibited, but the user should be aware that timing constraints may be affected by speed changes (in particular the Adaptive Partitioning Scheduler (APS) will not behave correctly in such a dynamically-changing environment).

REC-0009. If dynamic power management is used, the system designer SHOULD ensure that the minimum requirements of all safety functions are met by the lowest power state that the system may enter.

4.12 Floating-point operations

4.12.1 Emulated floating point

REC-0028. An application SHOULD NOT make use of emulated floating point (as distinct from hardware floating point) for computations that are safety-critical.

There are some extreme conditions exposed by William Kahan’s “paranoia” tests (first published in 1983) where the emulated floating point operations are inaccurate. In practice, these conditions are unlikely to be met during normal operation (and the inaccuracies are in the lower-order bits), but the user is warned to check any computations that are critical to the continued safe operation of the software.

Note that for the QOS itself, this recommendation is enforced. The QOS will not boot on a system that does not provide hardware floating point support. However, an application might still be compiled to cause it to emulate floating point instructions. In such a case, the application might be susceptible to the limitations of emulated floating point implementations.

4.13 Custom kernel calls

In some situations, a user may wish to provide target-specific functionality to 1 or more processes, and the implementation requires performing work at a privileged CPU level. QOS makes this possible by providing what's called a "custom kernel call".

A custom kernel call allows all processes to make a kernel call which will execute user-provided code at a privileged CPU level; Ring 0 on Intel, EL1 or EL2 on AArch64. To make use of this functionality, a user:

- defines a C-callable wrapper around the custom kernel call, and
- sets the system page's callout's field `custom` to point to their custom kernel call callout; i.e., the code to execute during the custom kernel call.

A custom kernel call callout can access or modify anything the kernel can.

RST-0129. If the custom kernel call callout is NOT NULL, then a system integrator SHALL perform a safety analysis of the implementation of the custom kernel call callout to the same ASIL/SIL level as the QOS.

RST-0130. If the custom kernel call callout is NOT NULL, then a system integrator SHALL perform a security analysis of the implementation of the custom kernel call callout.

REC-0070. A system integrator SHOULD ensure the following, if applicable:

- The functionality of the custom kernel call is restricted based on a process's type id, and
- If the process's type id does not permit execution of the custom kernel call, an error code of the implementer's choosing must be returned.

For more detail about the use of process's type id, consult *System Security Guide*, which can be found in QNX user documentation.

CHAPTER 5

QOS Libraries and Utilities

Contents

This chapter contains details on specific libraries and utilities delivered with the QOS.

Section 5.1 General

Section 5.2 The C runtime library (libc) containing most POSIX functions, C standard functions, and native QNX system calls.

Section 5.3 The libslog2 library for writing logs to a circular buffer managed by the slogger2 utility.

Section 5.4 The libfdt library used to manage flattened device tree data structures.

Section 5.5 The SMMUMAN utility for programming SMMU and IOMMU hardware.

Section 5.7 The C++ runtime library (libcxx).

Section 5.8 Potential issues with uncertified tools.

5.1 General

RST-0005. The QOS SHALL be accessed from application code only through interfaces published in the BlackBerry QNX public documentation.

5.2 libc

5.2.1 Certification scope

RST-0006. All developers creating applications to run on the QOS SHALL be aware of and SHALL follow the restrictions that the BlackBerry QNX library documentation (provided as part of the QNX SDP 7.1 installation) applies to the contexts in which different `libc` functions may be invoked.

The documentation published with the QOS for each function contains information that is necessary for the safe use of the function. In particular, not all functions are safe to invoke in all environments; for example, some functions should not be invoked from the context of an interrupt service routine or an instrumentation callout.

In the QOS library documentation, each library function is described individually, and each function description includes a "safety" table that indicates in what contexts the function may be safely used.

- The "Cancellation point" row of the "safety" table indicates whether the function can be safely used as a cancellation point. This means that if a thread cancellation is pending, the function may cause the thread to be terminated.
- The "Interrupt handler" row of the "safety" table indicates whether the function may be safely called from within an interrupt handler or from within an instrumentation callout. Violating this restriction may result in the QOS entering its design safe state.
- The "Signal handler" row of the "safety" table indicates whether the function may be safely called from within a signal handler. Violating this restriction may result in the death of the faulty application.
- The "Thread" row of the "safety" table indicates whether the function may be safely called from a multithreaded application. Violating this restriction may result in incorrect behavior of the function, possibly resulting in the death or failure of the faulty application.

5.2.2 Fortified system functions

Fortified functions are wrappers for `libc` and other functions that use GCC builtin functionality to detect out-of-bounds memory accesses by performing lightweight parameter validation at compile-time, runtime, or both.

QOS fortified system functions are designed to prevent the following scenarios:

- A function writes data outside the bounds of a destination object whose length can be determined at compile time.
- A function that accepts a variable number of arguments takes more arguments than the number actually specified by the caller.

The thoroughness of checks performed by fortified functions on objects varies based on the amount of information passed to the fortified functions. If no information is available for an object, then no extra checking is performed. Fortified functions should be enabled at compile time to avoid unnecessary runtime errors.

Fortified system functions perform checks during compilation and at runtime to detect and prevent out-of-bound memory writes. An out-of-bounds memory write can corrupt a program's state and lead to unintended behaviour. An attacker can exploit such a vulnerability to modify program behavior or execute arbitrary instructions. This section provides recommendations for configuring QOS to detect the instances of out-of-bounds memory writes.

REC-0051. The use of fortified system functions SHOULD be enabled at level 2, which is the highest fortification level supported.

To enable or disable the use of fortified system functions, define the `_FORTIFY_SOURCE` feature test macro with a value between 0 and 2 by any of the following means:

- the `#define` preprocessor directive (e.g. `#define _FORTIFY_SOURCE 2`), specified prior to any and all `#include` directives in a source file
- the `-D` compiler option (e.g. `-D_FORTIFY_SOURCE=2`), specified via the `CCFLAGS` and `CXXFLAGS` makefile variables
- the `-D` compiler option (e.g. `-D_FORTIFY_SOURCE=2`), specified via the `CCOPTS` and `CXXOPTS` environment variables prior to invoking the make utility

Level 0 disables fortified system functions and level 2 enables fortified system functions with more stringent parameter validation than level 1. For example, the compiler does not emit an error when the following code is compiled with a `_FORTIFY_SOURCE` setting of 1, but it does emit one when `_FORTIFY_SOURCE` is 2:

```
struct {
    char key[8];
    int  value;
} key_value_pair;

strcpy(key_value_pair.key, "too long");
```

The *strcpy()* tries to copy the string `"too long"` of length 8 bytes plus the terminating NULL character of 1 byte, thereby exceeding the size of `key` string. Level 2 performs checks to the bounds of individual structure members, not just the bounds of the parent object as in Level 1.

5.2.3 Forksafe mutexes

Some functionality of Forksafe mutexes has been moved out of the `libc` library into a separate `libforksafe_mutex` library for QOS 2.2.2. To link against the `libforksafe_mutex` library use `-l forksafe_mutex` option to qcc. However, the default implementation of forksafe mutexes in `libc` is still available but is merely a wrapper around standard pthread mutexes that are not Forksafe. The `libc` is usually included automatically. Use the `-l c` option to `qcc` if there is a need to explicitly link against the `libc` library.

The `libforksafe_mutex.so` binary is not safety certified and is only appropriate for use in QM applications. The following functions have moved to `libforksafe_mutex`:

- *forksafe_mutex_init()*
- *forksafe_mutex_destroy()*
- *forksafe_mutex_lock()*
- *forksafe_mutex_trylock()*
- *forksafe_mutex_unlock()*
- *forksafe_mutex_unlock_all()*
- *forksafe_mutex_cond_wait()*

5.2.4 Multi-threaded fork

REC-0029. The system programmer SHOULD NOT use POSIX *fork()* in a multithreaded process.

The POSIX function *fork()* can cause synchronization issues when used in a multithreaded process. The semantics around the *fork()* operation are complex, particularly when *fork()* is executed by a process with multiple threads.

The POSIX specification for the *fork()* function says that if a multi-threaded process invokes *fork()*, the child process will have only a single operational thread (corresponding to the thread in the parent that invoked *fork()*). Other threads in the parent process will not be recreated in the child, and any operations that were in progress by those other threads will be left in an unfinished and possibly inconsistent state in the child.

An application developer who depends upon the *fork()* function in a multi-threaded application must be aware of this characteristic of *fork()*.

5.3 libslog2

5.3.1 Overview

The `libslog2` library is the client side API for the `slogger2` logging utility bundled with SDP 7.1. Logging is an important function for development and debugging, but is also a necessary feature to support field monitoring.

The `libslog2` library sends one IPC message to the `slogger2` utility to register a log buffer. The `slogger2` utility must be running in order for the `libslog2` library to register a buffer correctly. All other `slogger2` API calls write log information into shared memory.

5.3.2 Using libslog2 in certified applications

The `slog2info` utility can be used to read the log buffers. However, this requires some mutual exclusion semantics between the client logging and `slog2info`. The use of `slog2info` during normal operation mode is discouraged. As an added precaution, the `libslog2` API aborts attempts to write to the buffer if it cannot gain write access after several attempts. The number of attempts before aborting is configurable by the user (flag *SLOG2_LIMIT_RETRIES*, see QNX public documentation).

Furthermore, the client API will continue to operate even if the `slogger2` server corrupts the log buffers.

RST-0053. Use of `libslog2` functions in a safety application SHALL be limited to those uses listed in Appendix F.

RST-0054. The `libslog2` library SHALL be used to register a buffer with the `slogger2` utility only during application startup.

On registering the `libslog2` library sends IPC messages to `slogger2` to initialize the log buffers. The `slogger2` server should not be relied upon for timely response during normal mode of operation.

REC-0037. The `slog2info` utility SHOULD NOT be used to read buffers during normal mode of operation.

REC-0038. Clients SHOULD set a timeout before calling *slog2_register()* and *slog2_set_verbosity()*.

Note that *slog2_register()* may send several blocking messages internally. See section 4.8.3 for details on setting a timeout when calling a function that potentially sends multiple messages.

REC-0039. The `slogger2` process SHOULD be registered with the server monitor with the SIGKILL signal.

REC-0041. A maximum number of retries to write an event to the buffer SHOULD be set for each log buffer.

The `libslog2` library prevents `slogger2` from creating a deadlock preventing a client from executing normally. In case the `slogger2` server is unresponsive, registration should occur during application initialization and the server monitor should be used to prevent `slogger2` from blocking the timeout of a registration request.

The use of `slog2info` during system operation is discouraged to avoid potential performance impacts. In the event of a failure, the logs may be read to capture field data once there is no longer a risk of interfering with the timing of the log-writer (i.e., once the application has reached a safe state).

5.4 libfdt

RST-0056. The `libfdt` library contains a QNX internal API and SHALL NOT be used by executables or libraries except those delivered by QNX.

The individual product documentation or reference BSPs of a product will include the `libfdt` library in the IFS as necessary.

5.5 SMMUMAN

The SMMUMAN component manages the IOMMU or SMMU for device bus accesses, see also section *4.5.2 DMA access*. The *smmuman* utility is included in the scope of certification and it:

- enables memory used by devices that deal with DMA traffic to be partitioned.
- restricts a resource manager to a particular partition, called *DMA caging*.

REC-0065. Non-safety (i.e. no SIL/ASIL) qualified resource managers SHOULD use a static configuration of the smmuman with a typed memory setup.

REC-0066. A non-safety certified resource manager SHOULD NOT use the *smmuman* API dynamically at runtime.

To prevent dynamic usage of the *smmuman* API by e.g. non-safety rated processes, security policies can be used to ensure this process cannot attach to the *smmuman* process, see section 6.3 Security policies.

Please refer to the SMMUMAN Safety Manual (QMS3341) and Hazard and Risk Analysis (QMS3333) for restrictions and recommendations on the use of SMMUMAN.

RST-0057. Users of the `smmuman` utility to manage SMMU/IOMMU hardware MUST comply with the *SMMUMAN Safety Manual*.

RST-0066. The `smmuman-safety` application MUST be spawned with the `POSIX_SPAWN_CRITICAL` attribute.

5.6 libm

The `libm` library implements math functions according to the C11 standard. Please refer to the Mathematics Library Safety Manual (QMS3359) and Hazard and Risk Analysis (QMS3358) for restrictions and recommendations on the use of `libm`.

RST-0058. Users of the `libm` library for mathematical functions MUST comply with the *Mathematics Library Safety Manual*.

5.7 libcxx

The `libcxx` Certified C++ Library is an implementation of the C++ standard library, targeting ISO/IEC 14882:[2011, 2014, 2017].

Please refer to QOS libcxx Certified C++ Library Safety Manual (QMS3570) and Hazard and Risk Analysis (*QMS3555*) for restrictions and recommendations on the use of `libcxx`.

RST-0108. Users of the `libcxx` Certified C++ Library MUST comply with the *libcxx Safety Manual*.

5.8 Uncertified tools

Some runtime utilities are mistakenly assumed to be certified. For example, both `aps` and `on` are useful development tools that should not be deployed in production certified systems.

Rather than using the `aps` utility, a program that calls the APS functions in `libc` to configure the schedule has to be written. The `aps` utility cannot be used in a production QOS system. Refer to the *Adaptive Partitioning User's Guide* for more information (http://www.qnx.com/developers/docs/7.1/#com.qnx.doc.adaptivepartitioning.userguide/topic/set_use_ap_crtepart_.html).

The `on` utility is convenient for development but not appropriate for managing permissions, security types etc. in a production QOS system.

For an alternative to the `on` functionality but also for existing shell scripts that use `sh`, a good start is to use the *posix_spawn()* API from the certified `libc` instead.
See http://www.qnx.com/developers/docs/7.1/#com.qnx.doc.neutrino.getting_started/topic/s1_procs_Starting_with_posix_spawn.html

CHAPTER 6

QOS and Security

Contents

This chapter contains information on safety aspects of QOS security features.

Section 6.1 Existence of malicious code on systems.

Section 6.2 Issues related to process permissions such as granting root privileges.

Section 6.3 Configuration of system wide security policies to ensure freedom from interference.

Section 6.5 Thread I/O privileges can violate freedom from interference.

Section 6.6 Require system level verification for pointer authentication.

Section 6.7 Require system level verification for Speculative Store Bypass Safe feature.

6.1 Malicious code

RST-0004. A system that deploys the QOS SHALL NOT contain malicious application code.

QOS cannot guarantee correct execution of the system in the presence of malicious code executing within the system. Integrity protection and security measures consistent with industry best practices must be applied to the system.

6.2 Process permissions

REC-0005. The system designer SHOULD make use of the flexibility provided by the QOS for placing access restrictions on servers (resource managers).

Applications that run with **root** privileges can overrule such access restrictions: see recommendations 10 and 11.

REC-0010. Applications SHOULD NOT execute with **root** privileges unless those privileges are required.

REC-0011. Applications SHOULD release **root** privileges as soon as those privileges are no longer required.

REC-0012. System developers SHOULD ensure that applications are granted only those permissions necessary to implement their functionality.

In the event of a failure, an application that has more privileges than it requires might interact or interfere with other applications in unexpected ways.

Be aware that the QOS implements a number of permission constraint mechanisms beyond those provided by the standard POSIX interface. In particular:

- the *procmgr abilities* mechanism allows the system designer to grant or restrict access to certain system functionalities independent of the process uid. This may allow a process to operate without root privilege where root would be required in a POSIX-only environment. It may also allow an unneeded privilege to be removed from a root process.
- the *security policy* functionality implemented with the `secpolcompile` and `secpolpush` utilities allows the system designer to impose additional process isolation constraints and also to centralize the assignment of procmgr abilities and other process privilege constraints, thus making it simpler to implement process isolation and privilege restriction correctly.

6.3 Security policies

Security policies specify rules to assign privileges to processes. Examples of privileges include setting the paths where channels are allowed to attach in the path space, defining which `procmgr` abilities to assign to processes, and controlling which processes can transition to another security types.

To establish a security policy, the following is needed:

- a well-defined set of processes to secure, and information about the interfaces between them
- labels for policy enforcement (type identifiers)
- rules for enforcement (process and channel attributes)

Policy development involves defining the rules in a text file according to a strict grammar, compiling them for integration into the system, and pushing the compiled policy to `procnto` to implement them. Once in place, these policy rules are enforced by the process manager rather than at the discretion of individual processes. The enforcement is type based. Rules associated with types in the policy determine the privileges given to processes that are run with those types.

The `secpolgenerate` utility assists users with security policy development and produces a file with security policy rules needed for observed system behaviors. The `secpolgenerate` utility also generates the file by monitoring a system's runtime and process behavior. During development, a system is often exercised in a way that processes gain extra privileges than necessary. Therefore, a security policy and an *unused* text files, generated by `secpolgenerate`, should be examined to ensure that processes have privileges as expected in the production environment. See Figure 6.1 for an overview of the QOS security policy workflow and consult QNX documentation (`http://www.qnx.com/developers/docs/7.1/#com.qnx.doc.security.system/topic/manual/tutorial.html`) for more detail about the correct configuration of the security policy.

Some restrictions are imposed when security policies are enforced on a target system. The following requirements describe the necessary measures in a system to keep QOS secure.

RST-0102. The binary `secpol.bin` SHALL be stored as read-only on the IFS.

The policy files and binaries may get corrupted in the storage media and can stop QOS from booting. Secure boot or another equivalent mechanism can be used to check the integrity of data when loading the IFS into memory.

RST-0103. The `secpolgenerate` utility SHALL never run in a production environment.

In the production environment, the system should have a static policy. `libsecpol.so` provides support for the secpol API that interacts with a real security policy when running a system securely. Similarly, `libsecpol-gen.so.1` provides support for the secpol API that interacts with a mock security policy during system development and is intentionally insecure. Therefore, avoid storing `libsecpol-gen.so` alongside `secpolgenerate` on the production IFS.

RST-0104. When configuring a security policy file, only those `libsecpol` functions described in the QOS documentation SHALL be used.

See QNX documentation for detailed `secpol_*` functions information.

RST-0105. When configuring security policy files, the following functions SHALL NOT be used:

- *secpol_find_blob()*
- *secpol_find_custom_blob()*
- *secpol_reset()*

- *secpol_find_entry()*
- *secpol_crc32()*

These functions are not qualified for use in application development. They are reserved for use only by tools, such as `secpolgenerate`, for generating policies.

RST-0106. The `procnto` option `-bl` SHALL be used to avoid errors caused by calls to *procmgr_ability()* when a process attempts to configure locked abilities.

Abilities can be modified within a process or outside the process using a security policy. If a process attempts to configure those abilities, which are locked by the security policy, the call to *procmgr_ability()* fails with unknown results. The `-bl` option on a production system avoids processes from failing when they attempt to configure locked abilities.

It's possible to miss or assign extra privileges to processes when configuring security policies. The following recommendations enable developers to configure a security policy with an optimal set of processes privileges.

REC-0056. The policy file generated by `secpolgenerate` SHOULD be reviewed and examined to ensure that the privileges specified for processes are assigned as expected.

The `secpolgenerate` utility generates a security policy file containing security rules obtained by observing the system in action to determine the set of privileges each process needs. A system is often exercised such that processes gain extra privileges than necessary. Therefore, a security policy file generated via `secpolgenerate` should be examined to ensure that processes have privileges as expected in the production environment. Additionally, the content of the unused file, also generated by `secpolgenerate`, should be consulted during testing to get an indication of extra privileges that might have been granted unintentionally.

REC-0057. When developing a security policy, ranges for process privileges SHOULD be specified where possible.

Some abilities may be specified over a range. Specifying ranges for abilities is useful to avoid granting more privilege than needed. For example, the following rule gives the type `server` the abilities that allow it to map physical memory over some range, switch to user IDs 4, 6, and 23, and create dynamic abilities:

```
allow server self:ability {
    mem_phys:0x1200000-0x24000000
    setuid:4,6,23
    able_create
};
```

REC-0058. In the security policy, security types that are permitted to spawn SHOULD be configured to change type automatically with limited privileges on spawn.

When a privileged process spawns a child, the child process may inherit the parent process's privileges even though it does not need them. To limit spawned processes from acquiring extra privileges, identify those security types that are permitted to spawn and configure them in the security policy to automatically change to a type with limited privileges on spawn.

REC-0059. If a security policy is enforced, resource managers SHOULD drop their privileges after initialization.

Resource managers generally require greater privileges during an early period of initialization. After initialization, they transition to another type as a means of dropping privileges. In a multi-threaded resource manager, there might be a scenario that the type will be switched before all initialization has been completed. This situation can cause the following problems:

- The process may sometimes fail to start properly because threads that are late in completing initialization have requests denied.
- To avoid the first problem, the process has to be given additional privileges long term that it requires only during start-up.

Therefore, ability dropping should be synchronized with activity across all threads.

6.4 Address Space Layout Randomization

Address space layout randomization (ASLR) varies the location of data and instructions each time a shared object library and executable[1] is loaded.

Due to randomization, ASLR helps protect against certain types of security vulnerabilities, such as Return-oriented Programming (ROP). ASLR does not affect the behaviour of correctly-written code, however, it may cause issues with code performing accesses beyond object boundaries (e.g. due to incorrect pointer use). The randomness in ASLR may make it difficult to use testing as a means of detecting out-of-bound accesses. Therefore, users should consider the benefits of ASLR versus their confidence in the correctness of their code to determine whether ASLR should be disabled.

6.5 I/O Privilege Levels

A thread running with I/O privilege means that the thread can:

- Disable interrupts on any CPU
- On Intel, directly interact with hardware in the I/O address space via the `in`, `out` opcodes
- Attach trace event handlers (which are invoked from high CPU privilege levels)
- Affect system-level registers (CR0, CR4, model-specific registers. EL1 registers)
- On Armv8-A systems, when running at Level 2, the thread runs at the same exception level as the kernel, and therefore may access and modify kernel data.

[1] If compiled to be a position-independent executable (PIE), which is the default

The QOS cannot provide protection against threads with I/O privileges and the onus is on the system safety analyst to guarantee freedom from interference (FFI).

RST-0127. An application thread that acquires I/O privileges with *ThreadCtl()* SHALL acquire the lowest I/O privilege level necessary.

REC-0068. An application thread that requires I/O privileges to perform work SHOULD only acquire I/O privileges when necessary. After performing work that requires I/O privileges, the thread SHOULD relinquish or reduce I/O privileges either by adjusting them with *ThreadCtl()*, or terminating.

REC-0069. The *ThreadCtl()* SHOULD be used with command flag `_NTO_TCTL_IO_LEVEL` instead of `_NTO_TCTL_IO` and `_NTO_TCTL_IO_PRIV`.

Both `_NTO_TCTL_IO` and `_NTO_TCTL_IO_PRIV` implicitly set the `_NTO_TCTL_IO_LEVEL_INHERIT` flag.

RST-0128. The *ThreadCtl()* command flag `_NTO_TCTL_IO_LEVEL_INHERIT` SHALL be set only when a thread with I/O privileges must create another thread requiring I/O privileges.

6.6 Pointer Authentication

Some verification effort has been performed to ensure that Armv8.3-A pointer authentication functionality is supported, however, no system level verification of this functionality has been performed.

RST-0131. If pointer authentication functionality is used in a system, then the system integrator SHALL take appropriate measures to ensure the functionality performs correctly, and

RST-0138. Since no safety analysis of pointer authentication functionality has been performed, the system integrator SHALL ensure that the system using pointer authentication functionality satisfies the requirements of the system's safety level.

6.7 Speculative Store Bypass Safe (FEAT_SSBS)

The Armv8-A FEAT_SSBS feature, which is controlled with the PSTATE.SSBS field, plus the use of speculation barriers, is meant to defeat side channels that might expose protected information to malicious actors.

Some verification effort has been performed to ensure that the correct configuration and use of PSTATE.SSBS and the speculation barriers is supported, however, no system level verification of this functionality has been performed.

RST-0132. If FEAT_SSBS functionality is used in a system, then the system integrator SHALL take appropriate measures to ensure the functionality performs correctly.

RST-0139. Since no safety analysis of FEAT_SSBS functionality has been performed, the system integrator SHALL ensure that the system using FEAT_SSBS functionality satisfies the requirements of the system's safety level.

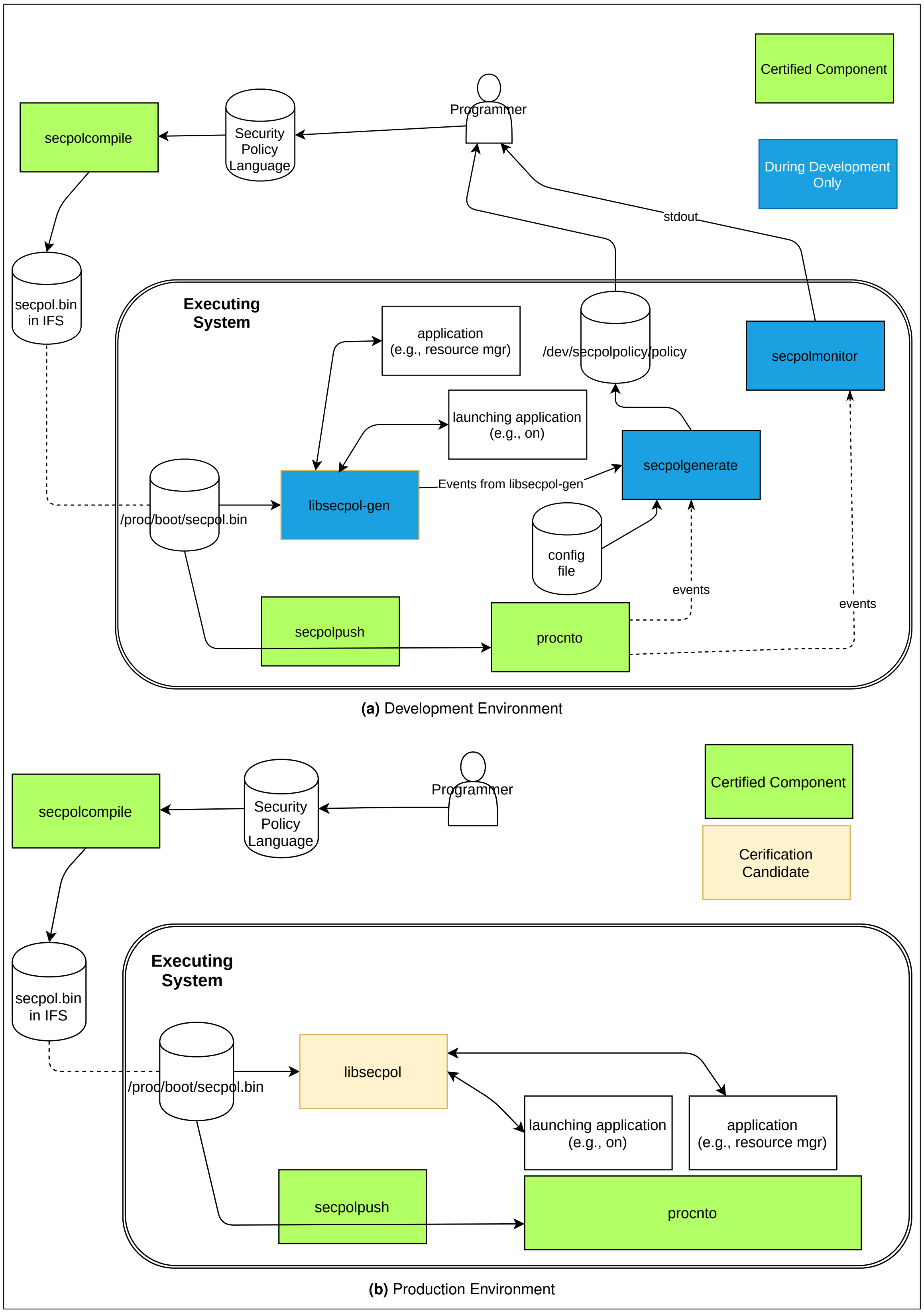

Figure 6.1: An Overview of the QOS Security Policy Workflow

CHAPTER 7

Handling Failures

Contents

This chapter contains information on handling failures in the QOS itself or its dependencies.

Section 7.1 Definition of the design safe state (DSS) of the QOS and how to detect and respond to it.

Section 7.2 Issues related to hardware failures.

Section 7.3 Checking that runtime components are not corrupted.

Section 7.4 Connecting with QM resource manager

Section 7.5 Side effects of terminating critical process

Section 7.6 Safe way of collecting crash dumps for terminated processes

7.1 Design Safe State

7.1.1 Handling unanticipated conditions

The QOS will sometimes meet conditions that it has not been designed to handle. These conditions may arise from:

- events external to the QOS. These include random hardware errors caused by cosmic rays, electromagnetic interference, or hardware degeneration caused by thermal effects. These events can change locations within memory or the processor in unpredictable ways.
- events within the QOS. These include combinations of events that cause the QOS to handle an event incorrectly.

When such a condition occurs, the QOS tries to move into its Design Safe State so that systems incorporating it can detect and handle the condition. It has to be accepted that the QOS itself is a program running on the hardware that may have been corrupted and, if the corruption is severe, it may not be possible for the QOS to reach its Design Safe State. It is assumed that the system-level design will accommodate this possibility by the use of external detection mechanisms (e.g., a watchdog timer).

7.1.2 Recovery

The system within which the QOS is operating must be able to handle its abrupt failure, which could occur at any time as a result of a power or hardware failure.

To simplify fault handling at the system level, the QOS does not make any effort to perform forward- or backward-recovery from an internally-detected error condition—it moves directly to its Design Safe State.

This removes the need to add complex code to the system to handle partial recovery or backed-out transactions.

7.1.3 How QOS enters its DSS

A situation that the QOS is not designed to handle causes it to crash cleanly in the manner described in the literature[1] as “Crash-Only”. This takes place as follows:

- The processor context is saved.
- Information about the type of exception and the saved register context is presented to an external interface for handling—in a development (rather than production) environment, this information is used by the kernel debugger or kernel dumper.
- The exception-processing code invokes the *shutdown()* routine.
- *shutdown()* outputs the values of certain important kernel variables, the register context at the time of the exception, and the bottom 128 bytes of the stack by invoking the *display_char()* kernel callout. This is a routine provided by the developer of the Board Support Package (BSP), and it may send the data to a serial port, store it in non-volatile memory, discard it, or take other appropriate action.
- Once the information has been written, the *shutdown()* code invokes the reboot kernel callout with an indication that this is an abnormal reboot request (i.e., not

[1] For example Hobbs, C., Becha, H., and Amyot, D., *Failure Semantics in a SOA Environment*, published in “Proceedings of the 2008 International MCETECH Conference on e-Technologies”, 2008, IEEE Computer Society, pp. 116—121.

generated as the result of an orderly shutdown). This callout is provided by the BSP developer and may take any appropriate action: typically forcing a reboot of the system.

This mechanism provides the application developer with a clearly-defined failure mode (crash-only) with some limited flexibility provided by the function of the debug and reboot callouts.

7.1.4 DSS Detection

RST-0012. The system within which the QOS is embedded SHALL detect the QOS moving to its Design Safe State and SHALL take the necessary action to preserve the safety of the system.

The QOS enters its Design Safe State only when it detects a condition which it has not been programmed to handle. An example might be a detected and unrecoverable corruption of the hardware on which it is running. Only a single Design Safe State is defined, and this is designed to be externally detectable.

7.1.5 Watchdog timers

REC-0017. A watchdog timer or other mechanism SHOULD be implemented at the system level in order to detect when the QOS has moved to its design safe state (DSS) or has become unresponsive in the case of silent failure.

The QOS may fail in a controlled manner by entering its DSS or fail silently without entering its design safe state and become unresponsive. System level mitigations should be designed to detect these scenarios.

7.2 Checking the integrity of hardware

RST-0087. System integrators SHALL implement any necessary mitigations against random hardware faults.

QOS does not detect most random hardware faults. Therefore, system designers are responsible for employing additional measures for mitigating hardware faults, such as using some defensive programming that can detect small portions of memory faults. Also, refer to Restriction 13 on system self-checking and hardware error reporting API.

RST-0013. System self-checking mechanisms SHALL be put in place to detect internal hardware faults by utilizing hardware fault detection mechanisms and fault modes provided by hardware manufacturers. Based on application's criticality level, detected faults/errors SHALL be reported to QOS by utilizing hardware error reporting API in order to maintain the required level of integrity for the application.

The detection of memory and processor errors in a timely manner is effectively impossible without hardware assistance (e.g., ECC Memory). The level of checking required depends on the level of acceptable failure for the particular device incorporating the QOS and would normally be determined from the system failure model. See

Reporting a Hardware Error on page 109 for the mechanism to be used to report the error.

Appendix D in ISO 26262-5:2018 and, in particular table D.1, provides a useful structure within which this type of analysis can be performed.

7.3 Checking the integrity of runtime environment

7.3.1 Binary integrity

REC-0003. The integrity of certified binaries SHOULD be checked at runtime.

Note that on a target system, files such as `procnto-smp-instr-safety` may generate an SHA512 value that differs from the value that was validated on the development host; this is due to how the `mkifs` utility processes ELF files (e.g., stripping out debug information) when it generates the IFS. The revision list provides values that can be checked at runtime with the `uname` and `use -i` utilities.

REC-0052. The integrity of the QOS Image Filesystem (IFS) SHOULD be checked when loading the system (e.g., Secure Boot)

QOS IFS stored in non-volatile memory may be corrupted or tampered with to modify data or executables, thus causing unsafe behaviour when booting the system. Secure boot or another equivalent mechanism can be used to check the integrity of data when loading IFS into memory.

7.3.2 System page integrity

REC-0055. The system page SHOULD be periodically checked for corruption.

QOS system page contains essential information about the system, including the system page's size, information about the hardware platform (including IRQs and the processor), the caches, and the location of kernel callouts in memory. The *syspage* entries are accessed with the *SYSPAGE_ENTRY()* macro. See the *C Library Reference* for further detail: `http://www.qnx.com/developers/docs/7.1/#com.qnx.doc.neutrino.lib_ref/topic/s/syspage_entry.html`.

The system developer can select the period for system page corruption checks based on the system's failure model. A brief description of system page fields that may be modified at runtime can be found in *QMS3310: Startup Module Design*.

Within *QMS3310: Startup Module Design*, there is an enumeration of important system page fields that do not change at runtime after the QOS initialization. If these fields are corrupted, they would likely have significant impact on the correct operation of the QOS.

Note the potential source of the QOS system page or kernel internal data structures corruption could be: SEU, ISR and a thread with I/O privileges at level 2. The memory data corruption due to SEU can be mitigated by the use of ECC memory. Similarly, ISTs can be used instead of ISRs (Recommendation 62). A user thread can only have pointer to system page as a read-only mapping. So even with I/O privileges at level 2 any user thread attempts to modify system page fields will receive SIGSEGV.

7.3.3 Checking binary versions at runtime

REC-0018. An application running on the QOS SHOULD confirm at application startup that the correct kernel is actually executing.

This can be confirmed by calling the *uname()* function from `libc` and checking the value of the *sysname*, *release*, and *version* fields to confirm that the installation process was correct and it is indeed the QOS that is executing.

Refer to the *QOS 2.2.2 Revision List* for the architecture-specific date associated with the `procnto-smp-instr-safety` binary. The revision list may be found by searching the certifying authority's web site at `https://fs-products.tuvasi.com/certificates?filter_prod=1&filter_apps=1&keywords=QNX&productcategory_id=1&x=0&y=0`

REC-0019. The version of the QOS binaries SHOULD be checked at runtime.

The version of a binary can be tested at runtime by executing the `use` utility. For example, the `libc.so` version can be tested as follows:

```
$ use -i libc
QNX_BUILDID=(GNU)5e23decac5ad4ac7537dfc3246fe389b
NAME=libc.so
DESCRIPTION=C runtime library
DATE=2019/07/11-19:25:47-EDT
```

and checking the `DATE` or the `QNX_BUILDID` associated with the binary.

`DATE` is preferred over the `QNX_BUILDID` if both are provided. Refer to the *QOS 2.2.2 Revision List* for the architecture-specific date associated with all certified binaries. The revision list may be found by searching the certifying authority's web site at `https://fs-products.tuvasi.com/certificates?filter_prod=1&filter_apps=1&keywords=QNX&productcategory_id=1&x=0&y=0`

7.3.4 Checking proncto command line option integrity

REC-0024. The integrity of the stored values of the `procnto` command-line options MAY be checked at runtime.

Refer to Appendix A.3 `procnto` command-line options on page 99 for a discussion of the recommendations relating to `procnto` command-line options.

Whether checking the integrity of the `procnto` command-line options at runtime is needed is determined by the failure model and dependability requirements of the application incorporating the QOS. If such checking is required, then see Appendix B Checking the Integrity of Command-Line Options on page 107.

7.4 Unresponsive resource managers and server monitor

7.4.1 Server monitor overview

The server monitor is used to identify when a server refuses to reply to an unblock pulse, thus preventing a client from exiting or entering a safe state.

The server monitor addresses an issue related to the implementation of correct fail-safe semantics in QNX systems.

Generally speaking, the fail-safe behavior of an application is to exit. The termination of a process is the safest and most direct method to ensure that no incorrect behavior occurs.

The QOS kernel will not allow a process to exit when it is reply blocked by a server. Clients may send an unblock pulse to servers to request that they be released. In exceptional circumstances, a faulty server may not respond to the unblock pulse request from the client, thus preventing the client from terminating.

The QOS kernel detects when an unblock pulse goes unanswered and notifies the server monitor. The server monitor may be configured to slay the offending server in order to allow the client to exit. Alternatively, if the server in question is used by other safety functions, it may be necessary to enter the QOS DSS (Design Safe State). A server monitor is configurable to take a different action for each server in the system.

There are several underlying assumptions for correct use of the server monitor.

1. The timeout threshold for which the kernel allows a server to respond to an unblock pulse is global. Therefore, the slowest response time of any server must be accounted for as well as any overhead associated with the QOS IPC itself.
2. A server should only be terminated if it can be demonstrated that the system can safely operate in a degraded state without the presence of the server. Otherwise, a QOS DSS must be initiated.
3. The server monitor must be spawned as a critical process. As a result, termination of the server-monitor process will trigger the QOS DSS.

7.4.2 Server monitor timing analysis

Figure 7.1 shows the interaction between a client, server, server monitor, and kernel. The client sets a timeout prior to initiating a *MsgSend()* with the server. When the transaction times out, the kernel sends an unblock pulse to the server. After the server fails to respond within the global timeout defined by the server monitor configuration, the kernel sends a notification pulse to the server monitor. The server monitor then executes the actions defined in the configuration.

Therefore, the worst-case response time for a client timeout is both the initial timeout value passed to the kernel in addition to the global timeout set for all servers to respond to an unblock pulse. In general, servers should be designed to respond to unblock pulses quickly and the additional timeout can be set quite aggressively.

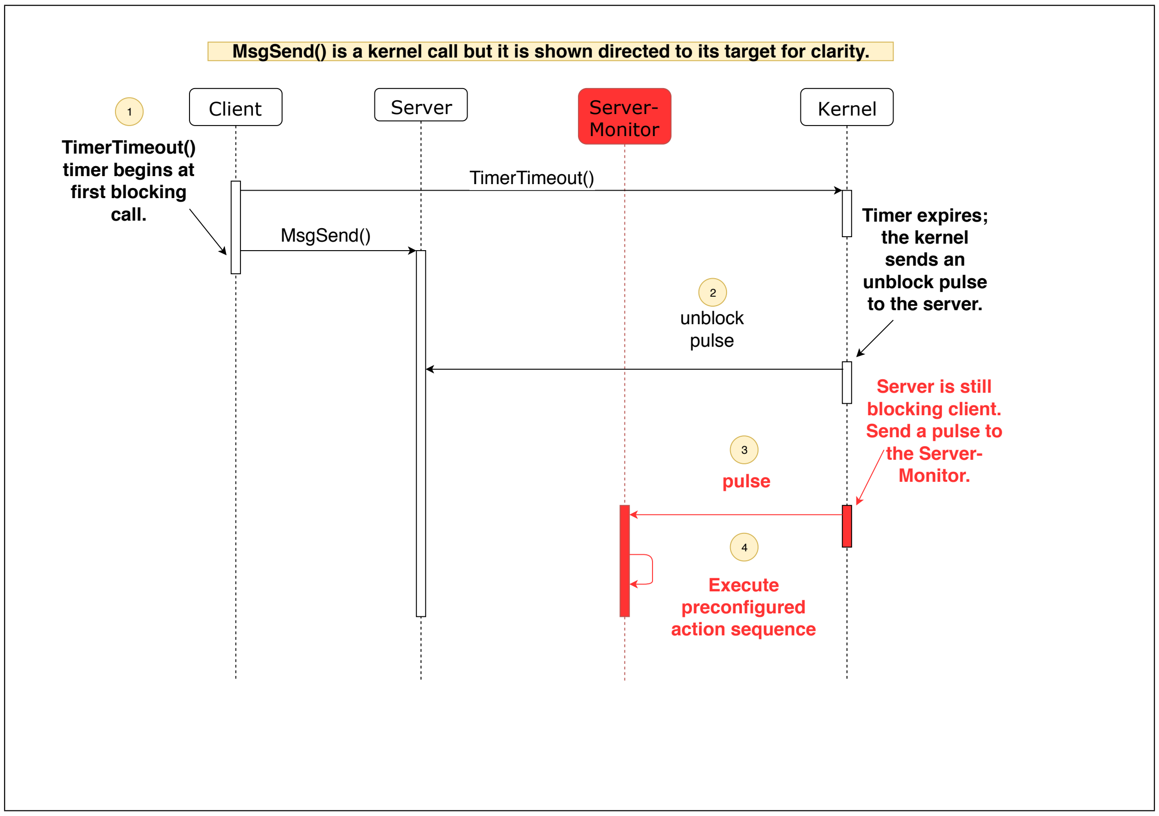
MsgSend() is a kernel call but it is shown directed to its target for clarity.
Client
Server
Server-Monitor
Kernel
1
TimerTimeout() timer begins at first blocking call.
TimerTimeout()
MsgSend()
Timer expires; the kernel sends an unblock pulse to the server.
2
unblock pulse
Server is still blocking client. Send a pulse to the Server-Monitor.
3
pulse
4
Execute preconfigured action sequence

Figure 7.1: Server monitor sequence diagram

7.4.3 Configuring the server monitor

When a client thread sends a message to a server, the client thread is suspended until the server responds. It is also possible for the termination of a client to be suspended while a client thread is sending a message to a server. If a client thread is reply-blocked when the client process is terminating, an UNBLOCK pulse is sent to the server. The client remains blocked until the server replies with *MsgReply()* or *MsgError()*. Only then will the kernel complete the client termination.

For a single-threaded server, the UNBLOCK pulse will not be seen until the client thread's message is replied to, so the worst-case delay is determined by the time it takes the server to respond.

In a multi-threaded server with all threads busy (and therefore not receiving messages), it will still be some time before receiving and acting on the UNBLOCK pulse.

REC-0033. The unblock pulse timeout value set for the server monitor SHOULD account for the expected worst case response time among all monitored servers and allow sufficient slack to account for IPC overhead.

To allow a client to exit or enter a safe state, an unblock pulse may be sent to a server with which it's communicating. If the server does not reply within a configurable timeout period, the server monitor is notified.

The server monitor is configured to allow a global response time for all monitored servers to respond to an unblock pulse, thus triggering a notification from the kernel to the server monitor.

The response time must account for the expected performance of all the monitored servers, as well as small amounts of jitter that may result due to concurrent kernel accesses from higher priority threads.

REC-0034. All high-priority or safety-critical clients requiring the server monitor service SHOULD be started after the server monitor.

The server monitor should be started as early as possible in the startup script to ensure that notification events from the kernel are not missed.

The server monitor uses the `libslog2` library in cooperation with the `slogger2` server to log events. There is a remote possibility that the server monitor could itself become blocked by the `slogger2` and fail to initialize correctly. The kernel provides a health check API for an external process to confirm that the server monitor is fully operational. Refer to the user documentation for the `server-monitor` utility for further details on the health check API.

REC-0035. Another process SHOULD verify that the server monitor is running correctly using the `_PROC_SERVER_MONITOR_ALIVE` message to `procmgr` during system initialization.

REC-0036. The system SHOULD be restarted or shut down if the return status of a `_PROC_SERVER_MONITOR_ALIVE` message is not `EOK`.

Note that the system should allow sufficient time for the server monitor to start or allow multiple attempts before shutting down.

RST-0067. The `server-monitor` application SHALL be spawned with the `POSIX_SPAWN_CRITICAL` attribute.

The server monitor could fail to perform an action upon detection of an unresponsive server if it executes at a lower priority than the server.

RST-0073. The `server-monitor` process SHALL run at a higher priority than all of the processes it is monitoring.

7.5 Critical processes

The availability of some services are critical to the correct operation of a system. If a critical process terminates, the QOS will enter its DSS.

System shutdowns must be handled properly when a system contains critical processes. Generally, a system shutdown consists of an ordered termination of all processes in the system prior to triggering a reboot via software or else powering off the device. Until the device is powered off, the kernel and any remaining processes are still active.

A critical process must not be terminated during normal system shutdown. It must be able to operate safely after all other processes are terminated as it will remain part of the minimal environment after a software initiated shutdown but prior to powering off the processor.

RST-0070. A system SHALL NOT terminate a critical process during normal shutdown.

RST-0071. The termination of any non-critical processes SHALL NOT cause any critical processes to terminate.

REC-0040. System integrators SHOULD spawn a process with the `SPAWN_CRITICAL` attribute if the system cannot operate safely after the process terminates or the process cannot be safely restarted.

For instance, `smmuman` is considered a critical process as spatial isolation can no longer be guaranteed after it terminates.

7.6 Crash dumps

7.6.1 Overview

QOS does not generate crash dumps. Instead, a process (the “dumper”) may register to be notified by QOS when a process abnormally terminates by attaching to the path /**proc**/**dumper**. When a process abnormally terminates, QOS sends a message (a "dumper message") to the dumper by opening /**proc**/**dumper** and writing the dumper message to the file. Although not required, it is common for a dumper to act on the information in the dumper message by using the /**proc**/***pid*** files to inspect and record the internal state of the abnormally terminating process in a crash dump.

The QOS Process Manager process (`pid 1`) is spawned with `uid 0` in QOS 2.2.2 and can access /**proc**/**dumper**. The Process Manager sends the dumper message to the dumper process. A thread within the Process Manager blocks on the dumper message while the dumper is generating the crash dump; this activity is expected to take anywhere from a few milliseconds to tens of minutes.

QOS supports union mounts, which means multiple resource managers can register to the same path, and this includes the dumper API path /**proc**/**dumper**. When registering a path, a resource manager may specify where it prefers to be placed in the search order (BEFORE others, AFTER others) when matching a file path to a resource manager. If the first resource manager reports an error when opening a file, QOS moves on to the next resource manager, repeating this until successfully opened, or all matching resource managers report failure.

QOS will not send a dumper message:

- if no dumper process is registered,
- before an abnormally terminating process is successfully loaded,
- if the abnormally terminating process is a critical process,

- when the Process Manager already has the maximum possible number of threads for a process (32767), or
- when the amount of available memory precludes creation of a thread by the Process Manager.

In the presence of a dumper process, if one or more processes abnormally terminate during system shutdown then it may result in longer shutdown times. Once the dumper process replies to the process manager, the /**proc**/***pid*** files associated with the abnormally terminating process are removed.

The scope of the dumper messages is limited to writes, i.e., messages of type `_IO_WRITE(64)` on /**proc**/**dumper** from the Process Manager (PID 1). Any other messages sent to file descriptors opened on /**proc**/**dumper** are out of scope, but not explicitly disallowed. A dumper message is sent to the dumper process within an individual `_IO_WRITE`. There are two possible formats of the dumper message. If the system resource limit of abnormally terminating process is set to `RLIM_INFINITY`, then the message format is

```
^(\d{1,10})$
```

where the message contains the process ID of the abnormally terminating process. otherwise the message format is

```
^(\d{1,10}) (\d{1,20})$}
```

where the first field is the process ID of the abnormally terminating process, and the second field is the value, in bytes, of process's `RLIMIT_CORE` system resource limit. Both values are represented in decimal (base 10) using ASCII. The dumper message does not include a terminating null character. For instance, if a terminating process has a process ID of `1380354` and `RLIMIT_CORE` is set to `256` bytes, then the dumper message is `1380354 256`. The maximum dumper message length can be of $10 + 1 + 20 = 31$ bytes excluding terminating null character. The dumper message size should be closely examined in case more data needs to be read with *MsgRead()*.

The dumper return value in response to a dumper message is used to reflect the status of the abnormally terminated process, and is detectable via *wait()*, *waitid()*, and *waitpid()*.

7.6.2 Writing a certified process dumper

RST-0112. A dumper process SHALL register the path /**proc**/**dumper** only after all preparations required to receive dumper messages are complete.

Once the dumper process has registered the path /**proc**/**dumper**, it may immediately begin to receive dumper messages. If there is still initialization work to be done, the dumper messages may not be handled correctly. The incorrect handling of dumper messages is even more likely if the dumper process is using resource manager thread pools.

RST-0113. A dumper process SHALL respond to all resource manager messages received on the path /**proc**/**dumper**.

If a dumper process does not respond to dumper messages then:

- Process Manager threads are consumed and not released, and
- it is likely to lead to a lack of resources, since the resources of the crashing process must be maintained until the QOS receives a reply to the dumper message.

RST-0114. A dumper process SHALL return either of the following return values:

- -1 if a core dump could not, or was not, successfully created, otherwise
- the size, in bytes, of the dumper message.

RST-0115. A dumper process SHALL register the path /**proc**/**dumper** as a file of type `_FTYPE_DUMPER`.

Using the `_FTYPE_DUMPER` file type for dumper messages eliminates the possibility of the process manager sending a dumper message to a non-dumper resource manager.

7.6.3 Certified dumper process

RST-0116. Any process that registers the path /**proc**/**dumper** with file type `_FTYPE_DUMPER` SHALL be safety-certified to the highest safety integrity level allocated to the software system.

RST-0117. Any process in a deployed system given access to the directory /**proc**/***pid*** and all files beneath SHALL be safety-certified to the highest safety integrity level allocated to the software system.

In order to create a crash dump, the dumper process needs the capability of accessing the /**proc**/***pid*** files, including /**proc**/***pid***/**as**. This allows a dumper process to violate spatial isolation between processes.

RST-0118. Any access to /**proc**/**dumper** SHALL be restricted to only those processes which require access to it.

Any process given permission to open /**proc**/**dumper** may send any kind of message to the dumper process. The dumper may not be able to handle these messages, or this could lead to unknown behavior.

CHAPTER 8

Performance Measurements and Tuning

Contents

This chapter discusses issues related to performance measurements, performance analysis, and performance tuning.

Section 8.1 issues related to modelling the performance of QOS based systems

Section 8.2 characterizing hardware impact on performance

Section 8.3 impact of hardware clock accuracy on software performance

Section 8.4 measuring time from within an application

Section 8.5 impact of context switches on resolution of measurements

Section 8.6 impact of the `-p` option for `procnto` on performance

Section 8.7 impact of high utilization on accuracy of software timers

Section 8.8 accurately measuring event ordering

8.1 Performance modelling

Tools are often used during the design process to assess system performance, and such tools are based on assumptions. BlackBerry QNX can assist with the validation of the assumptions made in such tools.

RST-0009. No assumption SHALL be made about the timing of specific library calls other than where such timing guarantees are given explicitly in published BlackBerry QNX documentation.

REC-0001. The assumptions made about inter-process communications within the QOS by design tools used to generate and validate high- and low-level designs SHOULD be identified and validated to ensure that they correctly reflect the actual operation of the QOS.

REC-0015. The system designer SHOULD ensure that the scheduling design of the system is kept simple enough that it is possible to model the scheduling behavior of the system with sufficient accuracy to meet design requirements.

The QOS provides a large number of different scheduling mechanisms such as priority-based scheduling, round-robin versus FIFO scheduling, APS (which itself has many different options), sporadic scheduling, and core affinity. Each of these mechanisms has many variables, and many of these mechanisms can be used together. Combining too many different mechanisms leads to a system that is difficult to understand.

8.2 Characterizing hardware

8.2.1 Clock jitter

REC-0016. The system designer SHOULD characterize the jitter of the system tick ISR under low and heavy load on any system to be developed.

Note that on Intel systems, the LAPIC on core 0 is the recommended hardware timer; the HPET has consistently shown significant jitter issues.

8.2.2 LSE atomics

REC-0053. The performance of LSE (Large System Extension) atomics SHOULD be compared with LLSC (Load-Link/Store-Conditional) atomics on any ARMv8.1-A system to be developed.

All the ARM application processor cores on a system might support LSE however, it is still important to profile the performance of both atomics (LSE vs LLSC) to make the optimum design decision for your system. If LLSC's performance is better on your system, then LSE optimized function SHOULD be disabled.

8.2.3 Number of processor cores

RST-0016. If the QOS is configured to use more than 8 CPU (application) cores, the system developer SHALL verify performance assumptions under the maximum application processing load.

The QOS is designed to provide the same functionality in different hardware environments, specifically on different processor architectures or with different numbers of cores. While the QOS will operate correctly in all supported environments, the performance of the QOS is difficult to characterize across all supported environments. Specifically, performance typically does not scale linearly with the number of processor cores.

8.3 Clock accuracy

RST-0019. If a system is deployed that depends upon a certain degree of clock accuracy to maintain a safety function, the system developer SHALL verify that the chosen clock is accurate enough to meet the requirements of the safety function in all expected environments and states, especially in the presence of Über-Authorities, if tickless is enabled, or High Resolution Timers are used.

RST-0020. If a system is deployed that makes extensive use of hardware or software timers, then the system developer SHALL verify the correct operation of their system during times of maximum timer fire frequency.

8.4 Monitoring execution times

The execution time for the process/thread can be determined by calling *clock_gettime()* or *ClockTime()*, with a process/thread CPU-time clock ID, e.g., passing `CLOCK_PROCESS_CPUTIME_ID` or `CLOCK_THREAD_CPUTIME_ID` to *clock_gettime()*.

If process CPU time in nanoseconds cannot be represented as a 64 bit value, then the query will result in QOS entering its DSS. The earliest time since system boot this can possibly happen (in years) can be calculated using the formula: `MAX_UINT64 / (NUM_PROCESSORS * 1000000000 * 3600 * 24 * 365.25)`

For instance, if a process that has 32 threads running on a 32 core system, then the earliest time to exceed 64 bits will be 18.26 years.

8.5 Measuring execution time with *pthread_getcpuclockid()*

Consider the following scenario:

1. Thread T_L is a thread that performs a computation (i.e., it is processor-intensive and does not make any kernel calls while performing the calculation). The computation was multiplying matrices.
2. Thread T_L is constrained to run on one particular core (CPU) on the SoC.
3. When it starts to execute, thread T_L reads the value of its own thread clock as returned by *pthread_getcpuclockid()*. Assume this has the value t_1.
4. Having recorded t_1, thread T_L performs its calculation — this takes a few hundreds of microseconds.

5. Thread T_L then reads the value of its own thread clock again (say t_2).
6. Thread T_L then calculates $t_2 - t_1$ and writes this value to a shared memory area. Thread T_L doesn't open the shared memory each time.
7. Thread T_L then waits on a timer of the order of several milliseconds and repeats from step 3.

The resolution of the measured execution time $t_2 - t_1$ can be affected by context switching and inter-processor interrupts (IPIs) when T_L is preempted. For example, Figure 8.1, shows how Thread B's execution time includes part of the context switch in and out of the thread and the handling of an IPI.

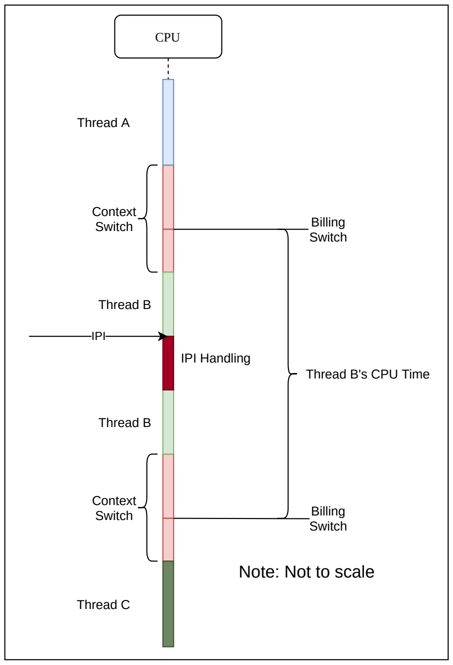

Figure 8.1: Illustration of time billing to thread

8.6 Kernel call latency, duration, and pre-emption

The QOS kernel supports a variety of mechanisms to avoid priority inversion. All of these mechanisms are intended to ensure that a high-priority thread cannot be blocked waiting for a lower-priority thread.

One such mechanism is the pre-emption of kernel calls. When a kernel call made by a lower priority thread is being processed on one processor core, and another, higher-priority, thread on another core makes a kernel call, then the first call will be pre-empted if and when it is safe to do so. The QOS ensures that kernel calls can be pre-empted only while kernel data structures are in a consistent state.

This mechanism minimizes the latency of the higher priority kernel call. It ensures that the higher priority kernel call is allowed to execute as soon as possible, even at the expense of delaying a lower-priority kernel call.

When a kernel call is pre-empted for another kernel call, the first kernel call is suspended until the second call completes. When the second call completes, the first cannot be allowed to continue from where it left off, as the second kernel call may have made changes to the state of the kernel. When it is allowed to resume, the first kernel call must be restarted to ensure that it correctly handles any state changes made while it was suspended.

Further, for the purpose of this discussion, interrupts that generate events to trigger interrupt service threads can be treated as kernel calls of the highest priority: the interrupt handler must enter the kernel to deliver or queue the generated interrupt event. Interrupt thread latency is minimized by allowing lower-priority kernel calls to be pre-empted to deliver high-priority interrupt events. (Interrupts generate events if they either have interrupt threads attached to them with *InterruptAttachEvent()*, or if they have interrupt service routines that are attached to them with *InterruptAttach()* and that return an event for the QOS to deliver.)

Note that the behavior of QOS kernel call pre-emption is similar to that of a typical priority-based scheduling algorithm, where a low-priority thread might never run if there are always higher-priority threads ready to run. The behavior of QOS kernel call pre-emption is unusual in the situation where a low-priority thread gets some processor time, but is interrupted too frequently by higher-priority events during a kernel call: in this circumstance it may never make progress even though it gets some processor time.

Because the generation of interrupt events must be handled in the kernel, any kernel call might be pre-empted and restarted by the generation of an interrupt event. It is also possible that a kernel call will not make any progress before it is pre-empted. If a kernel call is repeatedly and consistently pre-empted without making progress, it is possible that it might take a very long, potentially unbounded, time to complete.

Pre-emption of kernel calls by the QOS can be disabled through the use of the `-p` command line option. Refer to section A.3.2 `procnto` command-line options recommendations for an explanation of how to disable kernel call pre-emption.

In a system with kernel call pre-emption disabled, once a kernel call has begun, it is allowed to complete. In effect, this behavior is similar to priority boosting a low priority thread that holds a mutex when a higher-priority thread attempts to lock the mutex. The lower-priority thread's priority is boosted to that of the higher-priority thread until it completes the critical section of code. Similarly, a kernel call made by a low-priority thread "locks" the kernel. If another thread attempts to make a kernel call, the priority of the first thread is boosted so that the first kernel call completes without interruption.

It is important to understand that with kernel call pre-emption disabled, interrupt thread latency is also affected. While a low priority thread is executing a kernel call, interrupt events cannot be delivered and so maximum interrupt thread latency increases by the duration of the longest kernel call.

BlackBerry QNX does not recommend disabling kernel call pre-emption under normal circumstances, because doing so introduces a priority inversion between kernel calls and increases interrupt thread latency; high-priority kernel calls and interrupt event delivery must wait for lower-priority kernel calls to complete. However, as the frequency of interrupts or kernel calls increases, the maximum average latency of a kernel call made by a low priority thread also increases. A high-frequency interrupt with a regular period is particularly dangerous, as it could lead to a kernel call being constantly pre-empted and restarted without ever making progress.

Refer to restriction 17. BlackBerry QNX cannot characterize a specific interrupt frequency as dangerous, as it depends upon the characteristics of a specific system (including, but not limited to, the speed of the processor, the duration of any interrupt service routines attached to the high frequency interrupt, and the number of system resources such as channels consumed by the application). The system developer must ensure that applications behave correctly under maximum interrupt load.

The following summarizes this discussion:

- With kernel call pre-emption enabled (default behavior):
  - Higher priority kernel calls and events can pre-empt lower priority kernel calls. Priority inversion of kernel calls is avoided.
  - Kernel call and interrupt thread latencies are minimized
  - Lower priority kernel calls may be restarted and thus may take longer to complete
  - If event frequency is too high, in unusual circumstances, certain kernel calls may never complete.
- With kernel call pre-emption disabled (`-p` command-line option provided to `procnto`):
  - Once started, all kernel calls are allowed to run to completion
  - Entry to the kernel for a kernel call or to schedule an interrupt thread will be delayed by kernel calls of low-priority threads.
  - Maximum interrupt thread latency is increased by the duration of the longest kernel call.

8.7 Timer accuracy in high-utilization scenarios

- It should be noted that the POSIX standard, to which the QOS adheres, permits timers to expire *later* than the time for which they were set, but never *earlier*. The QOS follows this convention and, while a timer will never fire early, it may fire late.
- If an application has a large number of timers set to expire at the same time, then at most 50 of the timer events will actually trigger on time. With each timer interrupt, at most 50 timer events will be generated. Any expired timers left unfired during a tick are deferred to the next tick for handling.

This is designed to prevent too many system resources from being absorbed with timer events. Additional timers that were scheduled to expire but were not handled, will be handled on the next clock tick (subject to the same limitation of at most 50 timer events).

- The arrival of the ISR generated by the hardware timer that is used for the clock tick ISR will always be prone to jitter due to factors such as jitter in the hardware timer itself, and the prevention of clock tick ISR execution due to higher-priority interrupts. These delays and their jitter have been observed during testing. It is highly recommended that the delay and jitter be characterized on your platform with your configuration under realistic loads to understand real-world performance and implications.
- When significant jitter due to hardware timers was observed, it resulted in not only temporary but also long term drift of the monotonic clock —as well as other clocks derived from it—when compared to a separate and independent time source, e.g., *"ClockCycles"* (See "Segal's Law"). Making a comparison between two independent time sources can give the impression that a timer is firing interrupts late or early, but timers will fire interrupt after the fire time of their source clock.
- The effects on HRTs due to significant jitter in hardware timers can also be seen when the hardware timer ISR is prone to delay due to higher-priority interrupts.

8.8 Monotonic counter

A monotonic counter is provided for applications to make use of the “happened before” predicate. This counter may be accessed through the *clock_gettime()* `libc` function, using the `CLOCK_MONOTONIC` clock id.

The QOS guarantees that if a thread reads the monotonic counter at time t_1 and again at time t_2, where $t_1 < t_2$, then the value read at t_2 will never be less than that read at t_1.

Note without timecc kernel module (section 3.8), `CLOCK_MONOTONIC` is updated during a tick based on number of ticks, i.e., `SYSPAGE_ENTRY(qtime)->nsec` = number of ticks * *ClockPeriod*. However, when the timecc kernel module is in use, the `CLOCK_MONOTONIC` is updated during a tick as a function of *ClockCycles()*.

CHAPTER 9

Miscellaneous

RST-0023. An application SHALL NOT use atomic operations (including all mutex operations) in non-cacheable memory on AArch64-based systems.

The AArch64 architectures do not support atomic operations in non-cacheable memory. The results of such operations are undetermined. Any functionality, including locking or unlocking mutexes, that depend upon atomic operations will not operate correctly on non-cacheable memory.

RST-0119. Wide characters SHALL NOT be used in the code.

Example:

```
int main()
{
    wchar_t w  = L'A';
    wchar_t waname[] = L"geeksforgeeks";
}
```

REC-0075. In environments where there is significant preemption, and there are processes with a significant number of threads, channels, or connections, the system designer SHOULD perform the timing analysis to analyze whether the thread progression issue is applicable on their target platform.

On some older platforms, it has been observed in QOS that a thread calling *ConnectAttach()* will not return from this kernel call when:

- the thread is frequently preempted and
- the calling thread's process has a significant number of:
  - channels,
  - connections to channels (file descriptor (fd), or `SIDE_CHANNEL` connections) or,
  - connections to its channels, or

  - the above combined.

Similarly, it has not been observed, but it is theoretically possible on "slower" platforms that a thread calling *ConnectDetach()* may never return from *ConnectDetach()* in situations when there is frequent preemption during any call to *ConnectDetach()* and the calling thread's process has a significant number of threads. On the other hand, the call to *ChannelCreate()* and *ChannelDestroy()* returns successfully in all those situations described for *ConnectAttach()* and *ConnectDetach()*.

It is quite possible that a system designer will design a QOS system that may require more than 100 open channels and connections. To get a reasonable estimate of a time (T ms) that a kernel spent in a preemptable portion of a kernel call on a target machine, consider the preemptable and non-preemptable parts of the kernel call. The susceptibility to preemption is proportional to the length of time that a kernel call spends in the preemptable portion, which is proportional to the number of connections required to be scanned (N). Assuming that the time in the non-preemptable part of a kernel call is more or less constant, the two times can be estimated by timing the kernel call with no preemption when:

- N = 0 (time = non-preemptable)
- N = max (time = non-preemptable + worst case preemptable)

Preemptions can be taken into account from two sources: i) kernel calls from thread with priority greater than the priority of current thread in kernel call (in kernel, but not locked), and ii) interrupts. Also, see section *8.6 Kernel call latency, duration, and pre-emption*.

RST-0030. When a channel is disconnected, an application SHALL NOT rely on the ordering of the delivery of the pulses.

A kernel-generated disconnect pulse may be delivered before other pulses from the same client, even if the other pulses were of higher priority.

RST-0036. The system SHALL NOT execute the `mount_ifs` utility.

RST-0037. An application SHALL NOT modify opaque data structures created by the QOS.

This restriction is implicit in restriction 5, refer to page 51: all opaque data structures should be modified only through the APIs provided by the QOS.

RST-0038. The LD_BIND_NOW environment variable SHALL be set to prevent lazy function binding.

RST-0039. An application SHALL NOT create a robust mutex on a page of memory that was mapped into the application with the MAP_PHYS flag.

Mutexes with their robustness attribute set are called robust mutexes. They help recover the mutex if its owner terminates while holding it.

RST-0041. An application SHALL NOT use the -ad option to the `procnto` command line on AArch64 systems. Consult section A.3.2 `procnto` command-line options recommendations for details.

RST-0044. An application SHALL NOT set the *MALLOC_BAND_CONFIG_STR* environment variable.

RST-0045. An application SHALL NOT depend upon a thread that drops its own priority continuing to run if there are other ready threads of priority equal to the first thread's new priority.

POSIX specifies (see the *Rationale* section for the *pthread_setschedprio()* API at `http://pubs.opengroup.org/onlinepubs/9699919799/functions/pthread_setschedprio.html`) that if a thread is running at priority X while a second thread is ready at priority Y (where X $>$ Y), and the first thread drops its priority to Y using the *pthread_setschedprio()* API, that the first thread should continue to run. In the QOS, the first thread will be pre-empted and the second thread allowed to run.

RST-0046. Applications SHALL NOT depend upon an immediate timeout on a *MsgReceive()* call resetting the thread's APS partition.

When a thread in a resource manager receives a message from a client through a *MsgReceive()* call, it can inherit the client's priority and APS partition. If the next *MsgReceive()* call is given a non-immediate timeout and no message is received before the call times out, the server's thread will reset to its own APS partition. However, if there is no message waiting when a *MsgReceive()* call is made with an immediate timeout, the server thread's APS partition will not be reset.

RST-0047. Applications SHALL NOT depend upon the accuracy of scheduler time slicing when using the sporadic scheduling algorithm.

RST-0050. By default the clock period (the rate at which clock ticks arrive, in nanoseconds) will be 1 ms (1e+6 ns). If any other value is required, the value SHALL be established by calling *ClockPeriod()* with the desired value at boot time, as early as possible, and *ClockPeriod()* SHALL only be used thereafter for reading.

RST-0123. The startup binary SHALL ensure that the system page `qtime`, field *rr_interval_mul* value is set to 0.

rr_interval_mul represents the number of system ticks used by the round-robin scheduling interval. *rr_interval_mul* set to value 0 causes QOS to use the default value of 4.

RST-0061. A safety application SHALL NOT be built with both the `-g` and `-flto` command-line options provided simultaneously to the compiler and/or linker.

Link-time optimization (enabled with `-flto`) does not work well with the generation of debugging information (enabled with `-g`). The combination of these options is experimental and MUST be avoided when building a safety application.

RST-0062. A safety application SHALL NOT be built with both the `-O` *n* and `-ftree-vectorize` command-line options provided simultaneously to the compiler and/or linker.

RST-0033. An application SHALL NOT invoke the *dup2()* function while simultaneously creating other new file descriptors.

The *dup2()* function can erroneously return EBADF if another thread in the application creates a new file descriptor that matches the *dup2() newfd* parameter at the same time the *dup2()* function is executing.

RST-0040. An application SHALL NOT use the *SyncMutexEvent()* function on a mutex created in a page of memory that was mapped into the application with the `MAP_PHYS` flag.

Note that robust mutexes are created with the *pthread_mutexatttr_setrobust()* function. However, if the physical memory mapping is destroyed, the event attached to that particular mutex using *SyncMutexEvent()* will not be delivered.

RST-0094. The run state of processor cores SHALL NOT be changed. The functions *sysmgr_runstate()*, *sysmgr_runstate_dynamic()*, and *sysmgr_runstate-burst()* that are used for changing a core's run state SHALL NOT be used in certified applications.

RST-0095. The function *InterruptHookIdle2()*, which is used to attach an `_NTO_HOOK_IDLE` synthetic interrupt handler when a processor becomes idle, SHALL NOT be used in certified applications.

RST-0096. The *SchedCtl()* command `SCHED_CONFIGURE` SHALL NOT be used in certified applications.

RST-0097. The *LIBC_FATAL_STDERR_* environment variable SHALL be set to 1 in certified applications.

A failure in a libc function could result in error messages being sent to /**dev**/**tty**. This could be a result of stack overflow, and expose error messages to uncertified resource managers. If the *LIBC_FATAL_STDERR_* environment variable is defined then error messages are directed to *stderr* and eventually sent to `devc-pty`, but one at a time.

RST-0133. Applications SHALL NOT depend upon the accuracy of representation when converting between string and real number types.

Some rounding errors may occur during conversion. For example, the conversion of `2.99999999999999989e-01` with *strtod()* libc function returns `3.00000000000000044400e-01` in base 10 and `3fd3333333333334` in IEEE-754 floating point hex format.

APPENDIX A

Command-Line Options

Contents

A.1 Compiler/linker command-line options

The QOS is shipped with tools to build applications. This tool chain is the same tool chain used to build the binary portions of the QOS.

Refer to ISO 26262-8:2018 section 11 "Confidence in the use of software tools" for a description of the Tool Confidence Level concept that is used below.

A large variety of different command-line options may be specified to the compiler and linker. Some of these options control or modify the generation of the resulting binaries. It is not feasible to test that all combinations of all options work correctly.

When the tool chain is used to build the binary components of the QOS, different command-line options are specified for different components. These same command-line options have been subject to extensive verification by BlackBerry QNX with the binary components of QOS as well as third party test suites such as NPL and Perennial, that verify the semantics support of the compiler for the C and C++ languages.

This verification provides a significant part of the justification for BlackBerry QNX qualifying the tool chain to TCL-3 under ISO 26262-8 and to T-3 under IEC 61508-3. Specifying any other options that affect the code generation must be considered carefully, as BlackBerry QNX has not verified the correct operation of additional options.

A.1.1 Allowed command-line options

QOS provides different set of command-line options that developers MAY specify when compiling or linking safety applications. There are two set of command-line options:

1. `qcc`
2. `q++`

In the table below, the columns labeled *C* and *C++* refer to invocation of the back-end cc1 or cc1plus compiler. The column labeled *C* applies when `qcc` is invoked (with or without `-x c`), or when `q++` is invoked and `-x c` is applied. The column labeled *C++* applies when `q++` is invoked (with or without `-x c++`), or when `qcc` is invoked `-x c++` is applied.

Command-line Options	**C**	**C++**
`-V8.3.0,gcc_ntoaarch64le`	Yes	Yes
`-V8.3.0,gcc_ntox86_64`	Yes	Yes
`-V8.3.0,gcc_ntoaarch64le -march=armv8.2-a` (as a set)	Yes	Yes
`-fexceptions`	Yes	Yes
`-fno-exceptions`	Yes	Yes
`-fno-rtti`	Yes	Yes
`-Os`	Yes	Yes
`-O1`	Yes	Yes
`-O2`	Yes	Yes
`-fstrict-aliasing`	Yes	Yes
`-fno-strict-aliasing`	Yes	Yes
`-fcommon`	Yes	Yes
`-fno-common`	Yes	Yes
`-fomit-frame-pointer`	Yes	Yes
`-fno-omit-frame-pointer`	Yes	Yes
`-g`	Yes	Yes
`-no-pie`	Yes	Yes
`-fno-builtin`	Yes	Yes
`-fpermissive`	Yes	Yes
`-pedantic`	Yes	Yes
`-Wl,-s,-gc-sections -ffunction-sections -fdata-sections`	Yes	Yes
`-msse3 -mssse3 -msse4.1 -msse4.2 -fopenmp-simd`	Yes	Yes
`-D_FORTIFY_SOURCE=2 -DNDEBUG`	Yes	Yes
`-std=c++11 -D_QNX_SOURCE`	No	Yes
`-std=c++14 -D_QNX_SOURCE`	No	Yes
`-std=c++17 -D_QNX_SOURCE`	No	Yes
`-fstrict-overflow`	Yes	Yes
`-fno-strict-overflow`	Yes	Yes
`-funroll-all-loops`	Yes	Yes

Table A.1: Command-line Options (continued on the next page).

Command-line Options	C	C++
`-fwrapv`	Yes	Yes
`-faggressive-loop-optimizations`	Yes	Yes
`-fno-aggressive-loop-optimizations`	Yes	Yes
`-N{USER_DEFINED}`	Yes	Yes
`-fpic`	Yes	Yes
`-fstack-protector-strong`	Yes	Yes
`-Wl,-zrelro`	Yes	Yes
`-Wl,-znow`	Yes	Yes
`-shared`	Yes	Yes
`-I{USER_DEFINED}`	Yes	Yes
`-L{USER_DEFINED}`	Yes	Yes
`-x c`	Yes	Yes
`-x c++`	Yes	Yes
`-isystem <argument>`	Yes	Yes
`-nostdinc++`	No	Yes
`-nostdinc`	Yes	No
`-nostdlib`	Yes	No
`-D{USER_DEFINED}`	Yes	Yes
`-c`	Yes	Yes
`-Bstatic`	Yes	Yes
`-Bdynamic`	Yes	Yes
`-nostartfiles`	Yes	Yes
`-print-prog-name`	Yes	Yes
`-Wp{USER_DEFINED}`	Yes	Yes
`-U{USER_DEFINED}`	Yes	Yes
`-W{USER_DEFINED}`	Yes	Yes
`-Wc{USER_DEFINED}`	Yes	Yes
`-Wa{USER_DEFINED}`	Yes	Yes
`-v`	Yes	Yes
`-w`	Yes	Yes
`-dD`	Yes	Yes
`-o{USER_DEFINED}`	Yes	Yes
`-E`	Yes	Yes
`-l{USER_DEFINED}`	Yes	Yes

Table A.1: Command-line Options.

Where `{USER_DEFINED}` can be replaced with an appropriate argument to the compiler flag, e.g., `-I<dir>`, `-o<file>`, and `-Wa,<option>`. Note that signed integer overflow is undefined behavior in the C and C++ languages. Using the `-fwrapv` compiler option invokes a vendor extension that treats all signed integer overflow (or underflow) as a wrapper using two's complement representation, although it may impact some optimization operations. Further details of these interdependencies can be found in gcc documentation.

Note that system integrators are responsible for verifying the correctness of argument values where applicable.

Note that it is not required that all or even any of these options be specified when an application is built. Many of these options apply only to specific architectures and some are mutually inconsistent. These are simply the options that BlackBerry QNX has explicitly verified to generate correct code.

Note that BlackBerry QNX recommends that options that affect code generation, which are not listed above, be validated by the user according to either ISO 26262-8 or IEC 61508-3. BlackBerry QNX can provide services to assist users with the qualification of additional options. Further note that command line options that do not affect code generation (for example, those options that control diagnostic messages from the compiler) are considered safe.

A.1.2 Restricted command-line options

Use of high levels of optimization during application compilation is known to result in a variety of problems. Optimization levels of 2 or lower have been verified. Higher levels of optimization must be avoided.

Optimization levels are controlled by the `-O` *n* parameter passed to the compiler. Application developers SHOULD omit the `-O` option entirely, or specify `-O1` or `-O2`. Optimization levels higher than 2 (that is, `-O3` or higher) SHALL NOT be used.

A.2 `mkifs` command-line options

The `mkifs` tool is shipped with the QOS. A system developer will use `mkifs` to construct an *image file system*. The image file system is a binary image that defines the initial file system of a QOS system, including at least the startup component of the BSP, the QOS itself, and a set of commands that are executed during system initialization. The image file system may also include other files and directories. The contents of the image file system is controlled by a *build file* provided as input to the `mkifs` tool.

Consult the *SDP 7.1 Utilities Reference* for additional details describing how to use `mkifs`.

A.2.1 General `mkifs` Restrictions and Guidelines

When using mkifs to create boot images for safe systems, one very basic rule applies:

> Operative[1] information in the target IFS MUST NOT originate from anywhere in the host file system, apart from the build file used to create that IFS, and the content of the host files listed therein.

[1] "Operative" information would be anything that could affect the target's behavior; e.g., file content, file permissions. An example for non-operative information might be a file's modification time.

Especially, all items (files, directories, symbolic links) in the target IFS MUST be declared explicitly in the build file.

From this, a few "corollaries" can be derived:

- Each non-empty line in the build file (outside inline files) SHALL have any one of the following forms:

  ```
  '#' comment
  [attribute...]
  host_file_path
  target_file_path=host_file_path
  target_file_path=inline_file_spec
  [type=dir] target_dir_path
  [type=link] target_symlink_path=symlink_target_path
  [attribute...] host_file_path
  [attribute...] target_file_path=host_file_path
  [attribute...] target_file_path=inline_file_spec
  [type=dir attribute...] target_dir_path
  [type=link attribute...] target_symlink_path=symlink_target_path
  ```

  Section A.2.7 presents a formal syntax for mkifs build files used when creating boot images for safe systems.

- A host path SHALL NOT refer to anything other than a file or a symbolic link to an existing file. Especially, directories SHALL NOT be imported.
- The file/directory permissions (perms/dperms) as well as the user and group ID (uid/gid) of each item in the IFS SHALL be explicitly defined.

A.2.2 Restrictions On File Attributes

Note: Given the large value space for attribute parameters and the limited amount of non-runtime knowledge at image creation time the explicit declaration of attributes shall be verified as correct for the system configuration targeted. Best practices such as review or automated creation is highly encouraged.

A file attribute MUST NOT be specified more than once within the same set of attributes (i.e., within one pair of square brackets).

A.2.2.1 Mandatory File Attributes

The following line MUST be the first in each build file.

```
[-autolink -optional data=copy]
```

A.2.2.2 Allowed File Attributes

The file attributes listed below MAY be used in build files for safe systems:

```
cksum=...
```

```
+|-keeplinked
module=...
mtime=...
+|-raw
sha256=...
```

The `+raw` attribute MUST be used on all ELF files of type `ET_REL` (typically `.o` files). Without the `+raw` attribute, `mkifs` will attempt to link the object.

A.2.2.3 Restricted File Attributes

The file attributes listed below MAY be used in build files for safe systems, but restrictions apply either to the settings/values that are valid to use, or to the context in which they may occur: [2]

`autoso=`*`autoso_spec`*
- `autoso=`*`list`* used to discover required shared libraries SHOULD only be used during system integration. During testing, shared libraries discovered with `autoso=`*`list`* SHOULD be explicitly specified in the build file.
- `autoso=`*`add`* used to add all needed libraries to the Image Filesystem (IFS) SHOULD NOT be used.

`+|-compress`
: MUST be specified at most once per file. `-compress` SHOULD not be used (it is the default setting).

`dperms=`*`perm_spec`*
: All perm_spec attributes MUST take one of the following forms (expressed as a regular expression):
  - `u=s?r?w?x?,g=g?r?w?x?,o=t?r?w?x?`
  - `a=s?g?t?r?w?x?`
  - `0[0-7]?[0-7][0-7][0-7]`

  In other words, all bits of the file/directory permissions MUST be explicitly defined in the build file. [3]
  Note that while the *perm_spec* looks similar to permissions specification for the `chmod` command, they are not identical. Do not refer to `chmod` documentation when defining *perm_spec* values.

`gid=`*`id_spec`*
: All id_spec attributes MUST be decimal literals. Their value must be in the range [0..65536]. A value of '*' MUST NOT be used.

`image=`*`addr_space_spec`*
: MUST be specified at most once per file.

`perms=`*`perm_spec`*
: See "dperms".

`+|-script`

[2]Note that some of these (perms, dperms, uid, and gid) actually MUST be used in order to achieve defined target file attributes.
[3]Explicitly setting the "sticky" bit (symbolic 't', octal 01000) SHOULD be avoided. It is used to convey use-in-place (UIP) information within IFSs, and mkifs applies some fairly complex logic to calculate it (especially on ELF files, and on files marked executable).

SHOULD be specified at most once per file. `-script` SHOULD not be used (it is the default setting).

`type=`*`file_type`*
: SHOULD only be used to create directories (type=dir) or symbolic links (type=link).

`uid=`*`id_spec`*
: See "gid".

`virtual=`*`virtual_args`*
: MUST be used exactly once per file, in the "bootstrap" section. The target CPU type MUST be specified in the attribute.

A.2.2.4 Forbidden File Attributes

The file attributes listed in this section MUST NOT be used in build files for safe systems. Should a situation arise which appears to **require** the use of any of these attributes, the user should contact BlackBerry QNX.

```
+autolink
+|-big_pages
+|-bigendian
cd=...
chain=...
code=...
compress=...
filter=...
+|-followlink
linker=...
+optional
+|-page_align
pagesizes=...
phys_align=...
physical=...
prefix=...
ram=...
search=...
data=...
```

Note that the `compress=` attribute is distinct from the `+|-compress` attribute.

A.2.3 Bootstrap Restrictions

The bootstrap section (designated by the `virtual=` attribute) MUST NOT contain any global file attribute specifications. Any file attributes specified MUST apply to a single file on the same line.

The bootstrap section's "virtual" attribute SHALL explicitly specify the target CPU, e.g.,

```
[virtual=x86_64,bios.boot] .bootstrap={
```

```
...
}
```

A.2.4 Restrictions On Script Attributes

A.2.4.1 Required Script Attributes

The script attributes listed below MUST be used in build files for safe systems:

```
pri=...
```

The priority of a command in the startup script SHALL be specified using the `pri=` attribute. This attribute may be given on a line by itself within the script (in which case it applies to all subsequent lines), or it may be given on a line with a command (in which case it applies to the one command on that line).

A.2.4.2 Allowed Script Attributes

The script attributes listed below MAY be used in build files for safe systems:

```
argv0=...
cpu=...
+|-session
sched_aps=...
```

A.2.4.3 Forbidden Script Attributes

The script attributes listed in this section MUST NOT be used in build files for safe systems. Should a situation arise which appears to **require** the use of any of these attributes, the user should contact BlackBerry QNX.

```
+|-external
```

A.2.5 Restrictions On The Command Line

Certain restrictions apply to the command line used to invoke `mkifs`.

The command line MUST have the form

```
mkifs [allowed-option...] in-file out-file
```

or

```
mkxfs -tifs [allowed-option...] in-file out-file
```

A.2.5.1 Allowed Options

The command-line options listed below MAY be used with `mkifs` when building IFS images for safe systems:

`-?`
: Display some help information.

`-a suffix`

Append a suffix to symbol files generated via `+keeplinked`.

`-o directory`
: Output directory for all non-IFS permanent build artifacts.

`-r rootdir`
: Search default paths in *rootdir* before the default location.

`-s section`
: Don't strip the named *section* from ELF executables.

`-v`
: Increase verbosity.

A.2.5.2 Forbidden Options

The command-line options listed in this section MUST NOT be used with `mkifs` when building IFS images for safe systems. Should a situation arise which appears to **require** the use of any of these options, the user should contact BlackBerry QNX.

`-l` *inputline*
: Process inputline before interpretation of the buildfile begins.

`-n[n]`
: Force the modification times of all `-nn` or all inline `-n` files to be 0. If you want to create images with constant time stamps, use a global `[mtime=...]` attribute.

`-p` *patchfile*
: Apply patching instructions from this file.

A.2.6 Further Restrictions

A.2.6.1 Characters

The character set used in `mkifs` build files SHOULD be restricted to 7-bit ASCII. A build file SHALL NOT contain any non-printable characters, with the exception of LF (linefeed), HT (horizontal tab), and CR (carriage return). CR SHOULD NOT be used, unless operating purely under Windows.

The backslash character ('\') SHALL NOT be used other than as an escape prefix for one of the characters '\', '$', or '"'. The backslash character MUST NOT be used at the end of a line to achieve line continuation.

A.2.6.2 Environment Variables

In `mkifs` build files, the user can assign environment variables that should be passed to the programs specified in the bootstrap and script sections. These variables can either be applied 'locally' (if a command line follows the variable assignment), or be set globally (if no command follows). Note that these are not applied to mkifs's own environment.

Technically, environment variables (both of mkifs and defined in the build file) can be referenced (using the '${...}' syntax) in any place where text (e.g., attributes, file names, commands) is expected. Yet, restrictions apply if a boot image for safe systems is to be generated.

References to environment variables SHOULD refer only to variables defined within the build file; exceptions are "${MKIFS_PATH}" and "${QNX_TARGET}". Other `mkifs` environment variables MAY be used if necessary (e.g., to properly address files in a BSP).

The names of all environment variables defined in the build file SHALL have the form of valid C identifiers. They MUST NOT contain escaped characters ('\'...), quoted text ('"..."'), or variable references ('${...}').

References to environment variables SHOULD NOT be used on the right-hand side of environment-variable assignments.

A.2.6.3 Plain Keywords

Attribute and modifier names MUST be given in plain, unquoted text. They MUST NOT contain variable references.

A.2.6.4 Erroneous Artifacts

Even if `mkifs` terminates due to an error, it may leave behind an "IFS" output file. This file will NOT be a proper IFS. Output files from `mkifs` runs which terminated with an error MUST NOT be used.

A.2.7 Buildfile Grammar

Build files for safe systems SHALL be produced by the syntax shown below. Note that, for the sake of readability, the grammar is not formally complete.

```
build_file : head compress image bootstrap script body
          ;

head : space '[-autolink -optional data=copy]' LF
     ;

compress : space
         | space '[+compress]' LF
         ;

image : space
      | space '[image=' ADDR_SPACE_SPEC ']' LF
      ;

bootstrap : space '[virtual="' virtual_args '"] ' FILE_NAME '={' LF bootstrap_text '}' LF
          ;

virtual_args : CPU ',' BOOTER_NAME
             | CPU ',' BOOTER_NAME ' ' FILTER_ARGS
             ;

bootstrap_text : startup procnto space
```

```
               ;

startup : space startup
        | opt_fattr_spec opt_env STARTUP_COMMAND_LINE LF
        ;

procnto : space procnto
        | opt_fattr_spec opt_env PROCNTO_COMMAND_LINE LF
        ;

script : space '[+script] ' FILE_NAME '={' LF script_text '}' LF
       ;

script_text :
            | script_text space
            | script_text global_env
            | script_text global_modifiers
            | script_text opt_modifiers opt_env COMMAND_LINE LF
            ;

opt_env :
        | env_assign_list
        ;

global_env : env_assign_list LF
           ;

env_assign_list : env_assign_list env_assign
                ;

env_assign : ENV_VAR_NAME '=' ENV_VAR_VALUE
           ;

global_modifiers : modifier_spec LF
                 ;

opt_modifiers :
              | modifier_spec
              ;

modifier_spec : '[' modifier_list ']'
              ;

modifier_list : modifier
              | modifier_list modifier
              ;

modifier : 'argv0=' VALUE
         | 'cpu=' NUMBER
         | 'pri=' PRIORITY [SCHED_POLICY]
         | [+-] 'session'
         | 'sched_aps=' PARTITION_NAME
         ;

body :
     | body space
     | body global_fattr_spec
     | body file_spec
     | body dir_spec
     | body link_spec
     ;

space :
      | space LF
      | space '#' COMMENT_TEXT LF

file_content : host_file_path
             | inline_file_spec
             ;

inline_file_spec : '{' LF INLINE_FILE_CONTENT '}'
                 ;

global_fattr_spec : fattr_spec LF
                  ;

file_spec : opt_fattr_spec host_file_path LF
          | opt_fattr_spec TARGET_PATH '=' file_content LF
          ;

dir_spec : '[type=dir ' opt_fattr_list '] ' TARGET_PATH LF
         ;

link_spec : '[type=link ' opt_fattr_list '] ' TARGET_PATH '=' symlink_target_path LF
          ;

host_file_path : PATH_TO_FILE_ON_HOST
               ;

symlink_target_path : TARGET_PATH
                    ;

opt_fattr_spec :
               | fattr_spec
               ;

opt_fattr_list :
               | fattr_list
               ;

fattr_spec :  '[' fattr_list ']'
           ;

fattr_list : file_attribute
           | fattr_list ' ' file_attribute
           ;

file_attribute : 'cksum=' CHECKSUM(uint)
               | 'dperms=' PERM_SPEC
               | 'gid=' ID_SPEC(uint16)
               | [+-] 'keeplinked'
               | 'module=' MODULE_NAME
               | 'mtime=' TIME_SPEC
               | 'perms=' PERM_SPEC
               | [+-] 'raw'
               | 'sha256=' HEX_STRING[64]
               | 'uid=' ID_SPEC(uint16)
               ;
```

A.3 `procnto` command-line options

When the QOS kernel (`procnto`) is executed, various command-line parameters can be passed to it. These command-line parameters affect the behavior of the QOS in different ways.

The default values for some of these command-line parameters differ between the safety variant of `procnto` supplied with the QOS and the standard variant of `procnto` supplied with the non-safety release of QNX Neutrino. In all cases, the recommended setting of the command-line parameter for the QOS is the default.

The purpose of this appendix is to list the behaviors controlled by command-line parameters that are specifically recommended or not recommended for use with the QOS.

When an option is not mentioned below, it can be assumed that BlackBerry QNX makes no recommendation for or against its use.

For full descriptions of all command-line options to `procnto`, consult *Installing and Using the QNX OS for Safety 2.2.2*.

A.3.1 Specifying the `procnto` command-line options

The command-line options for `procnto` are specified in the configuration file (known as the “buildfile”) passed to the `mkifs` utility. This configuration file contains an ordered list of bootstrap executables which must start with the `startup-*` executable and end with `procnto`. This is fully defined in the *Utilities Reference* for `mkifs`.

A.3.2 `procnto` command-line options recommendations

The command-line options listed below provide specific recommendations for their use when starting `procnto`. The BlackBerry QNX user documentation should be consulted to determine how the multiple *-m* options are actually encoded on the command line.

Note that in all cases, the default configuration for the QOS matches these recommendations.

Option: `-ae` on ARM and AARCH64 systems

Effect: Enable unaligned memory access support.

Reason for recommendation: The QOS requires support for unaligned memory access on arm and aarch64 systems.

Recommendation: Leave unaligned accesses enabled (as is the default). DO NOT specify -ad to disable support of unaligned memory accesses.

Option: `-p`

Effect: Prohibit the pre-emption of most kernel calls.

Reason for recommendation: It may be desirable to reduce the maximum duration of a kernel call, at the expense of increasing the mean latency to enter the kernel and the mean and maximum latency of scheduling interrupt threads.

Consequences: With kernel call pre-emption disabled, a kernel call for a high priority thread may be blocked while a low priority thread's kernel call completes.

Recommendation: BlackBerry QNX does not recommend a particular value for this command-line option. The application architect should consult section *8.6 Kernel call latency, duration, and pre-emption* and determine whether it is appropriate to set this option. The default behavior of the QOS is to enable kernel call pre-emption.

Option: `-mL`

Effect: Map memory immediately so that a page fault will never occur. Ensure that virtual to physical memory mappings are constant.

Reason for recommendation: This option provides extra predictability and consistency to the application, both in terms of available of resources and execution time.

Consequences: Process startup time and memory consumption will increase, as all code and data must be loaded at startup time instead of being loaded as needed.

Recommendation: Leave memory mapping immediate (as is the default).

Option: `-m~x`

Effect: On hardware architectures that support the option, this option makes stack memory non-executable.

Reason for recommendation: This protects an application from accidental or malign attempts to inject insecure code into it and thereby change its behavior unpredictably. With this option, if an attempt is made to execute a program pushed onto the stack, then an exception will occur and the application will be terminated.

Recommendation: Leave stack non-executable (as is the default).

Option: `-F` *number*

Effect: Set the default hard rlimit value for the maximum number of open file descriptors.

Reason for recommendation: This option allows anomalous behavior of the system (e.g., a rogue task consuming an unexpected number of file descriptors) to be detected and the appropriate action to be taken. During testing, this may point to a design or system engineering flaw; during field execution it might lead to the system moving to its Design Safe State.

Recommendation: Analyze your application fd usage and consider changing the default value of 1000, and consider reducing the rlimit value for specific applications.

Option: `-H` *size*

Effect: Set the initial `procnto` heap size.

Reason for recommendation: Allocating a heap at startup that is sufficiently large to satisfy all of `procnto`'s subsequent needs guarantees that `procnto` will not need to allocate additional memory dynamically, and thus guarantees that `procnto` will not be subject to a situation where a necessary memory allocation fails because no memory is available.

Recommendation: Analyze `procnto`'s heap usage while running your application and consider changing the default value of 64KB to prevent `procnto` from needing to grow its stack.

Option: `-s`

Effect: Enforce safe handling of mutexes shared between processes.

Reason for recommendation: If this option is not set, unrelated processes might interfere through corrupted mutexes. One process might corrupt a mutex and, on attempting to lock the corrupted mutex, cause the priority of an unrelated process to be boosted.

Consequences: Operations on shared mutexes will be slower.

Recommendation: Leave handling of shared mutexes safe (as is the default).

Option: `-n`

Effect: Use nonlazy stack allocation.

Reason for recommendation: Lazy stack allocation may cause a delay in the detection of resource exhaustion.

Consequences: The system may fail if resources are exhausted.

Recommendation: Use the -n option.

Option: `-t`

Effect: Allocate software timers when a thread is created. The thread will fail to be created if there is insufficient memory for the software timer.

Reason for recommendation: There may be insufficient memory when a function is called that requires the software timer (e.g., *usleep()*).

Consequences: Developers may assume that a function such as *usleep()* cannot fail, and a program may operate with incorrect timing.

Recommendation: Enable pre-allocation of the software timer with `-t` to avoid the possibility of insufficient memory.

Option: `-bl`

Effect: Calls to `procmgr_ability()` that specify locked abilities will not fail.

Consequences: It prevents processes from failing because abilities are being controlled outside of a process (e.g., using a security policy).

Recommendation: The `-bl` option is the default behavior and should not be removed.

Option: `-c`

Effect: Sets the maximum offset (difference) tolerated in the clock cycles of the QOS system's processor cores.

Consequences: If the measured maximum offset is greater than 1 microsecond, then the kernel will compare the offset to the value provided for the `-c` option. If the maximum offset is greater, then the kernel deliberately crashes and produces a kernel dump.

Recommendation: The `-c` option should be used to set a maximum CPU clock offset. Note that the `-c` option will be ignored if the `QTIME_FLAG_GLOBAL_CLOCKCYCLES` is set in the flags for the qtime section of the system page.

Option: `-fa`

Effect: Pre-allocate an FPU context structure for every new thread.

Consequences: The FPU context will be allocated dynamically as needed if this option is not set.

Recommendation: The `-fa` option is the default behavior. The `-~fa` option should not be used to disable this feature.

A.3.3 `procnto` command-line options restrictions

Option: `-U`

Effect: By default in QOS, a server can deliver only those events that are registered with the client connection. This option enables the use of unregistered events.

Consequences: A server can send any event to any client connected to the server. This may lead to undefined behavior, for instance, a server can modify the memory of a client process.

Restriction: The `-U` option SHALL NOT be used in a safety system. If this feature is required, then enable it only for per-client connection with a server. (See Recommendation 43).

A.4 strip command-line options

The QOS is shipped with the command-line `strip` utility (`x86_64-pc-nto-qnx7.1.0-ar-2.32` and `arch64-unknown-nto-qnx7.1.0-ar-2.32 variants`). Refer to ISO 26262-8 section 11 "Confidence in the use of software tools" for a description of the Tool Confidence Level concept that is used below.

A large variety of different command-line options may be specified to the `strip` command. Some of these options control or modify the generation of the resulting binaries. It is not feasible to test that all combinations of all options work correctly. The command-line options detailed below have been subject to extensive verification by BlackBerry QNX.

This verification provides a significant part of the justification for BlackBerry QNX qualifying the tool chain to TCL-3 under ISO 26262-8 and to T-3 under IEC 61508-3. Specifying any other options that affect the code generation must be considered carefully, as BlackBerry QNX has not verified the correct operation of additional options.

A.4.1 Allowed command-line options

A developer MAY specify the following options when stripping files in safety applications:

`-V`
`-s`

These are the options that BlackBerry QNX has explicitly verified.

Note that BlackBerry QNX recommends that options that affect the resultant binary, which are not listed above, be validated by the user according to either ISO 26262-8 or IEC 61508-3. BlackBerry QNX can provide services to assist users with the qualification of additional options. Further note that command-line options that do not affect the resultant binary are considered safe.

A.4.2 Restrictions

In order to limit the field of testability, Table A.2 lists the restrictions placed on the file which is being operated on by the `strip` command:

Input	Restriction	Notes
file type	ELF binary, archive, shared library, static library	Target file cannot have a mixture of ELF architectures and/or non-ELF files
file size	100MB	Maximum size of target file
file number	10,000	Maximum number of files in the target file
file format	5000 ELF sections (up to 500 different types)	Maximum number of ELF sections
file names	64 Characters	Maximum filename length (not including path)
file path	256 Characters	Maximum path length (Note: The limit on Windows is 292 for both file name and path)
link number	0-6 depth	Maximum number of symbolic link redirections (Not supported on Windows)

Table A.2: Restrictions on the strip utility

A.5 ar command-line options

The QOS is shipped with the command-line `ar` utility (`x86_64-pc-nto-qnx7.1.0-ar-2.32` and `arch64-unknown-nto-qnx7.1.0-ar-2.32 variants`). Refer to ISO 26262-8 section 11 "Confidence in the use of software tools" for a description of the Tool Confidence Level concept that is used below.

A large variety of different command-line options may be specified to the `ar` command. Some of these options control or modify the generation of the resulting binaries. It is not feasible to test that all combinations of all options work correctly. The command-line options detailed below have been subject to extensive verification by BlackBerry QNX.

This verification provides a significant part of the justification for BlackBerry QNX qualifying the tool chain to TCL-3 under ISO 26262-8 and to T-3 under IEC 61508-3. Specifying any other options that affect the code generation must be considered carefully, as BlackBerry QNX has not verified the correct operation of additional options.

A.5.1 Allowed command-line options

A developer MAY specify the following options when archiving files in safety applications:

```
-r
-V
-c
```

These are simply the options that BlackBerry QNX has explicitly verified.

Note that BlackBerry QNX recommends that options that affect the resultant binary, which are not listed above, be validated by the user according to either ISO 26262-8 or IEC 61508-3. BlackBerry QNX can provide services to assist users with the qualification of additional options. Further note that command-line options that do not affect the resultant binary are considered safe.

Input	Restriction	Notes
file type	ELF binary, archive, shared library, static library	Archives cannot have a mixture of ELF architectures and/or non-ELF files
file size	100MB	Maximum size of file to be archived
file number	10,000	Maximum number of files in the archive
file format	5000 ELF sections (up to 500 different types)	Maximum number of ELF sections
file names	64 characters	Maximum filename length (not including path)
file path	256 characters	Maximum path length (Note:Windows limit is 292 for both file name and path)
link number	0-6 depth	Maximum number of symbolic link redirections (Not supported on Windows)
@file	not supported	

Table A.3: Restrictions on the ar utility

A.5.2 Restrictions

In order to limit the field of testability, Table A.3 lists the restrictions placed on the file which is being operated on by the `ar` command:

A.6 objcopy command-line options

The QOS is shipped with the command-line `objcopy` utility (`x86_64-pc-nto-qnx7.1.0-objcopy-2.32` and `arch64-unknown-nto-qnx7.1.0-objcopy-2.32`). Refer to ISO 26262-8 section 11 "Confidence in the use of software tools" for a description of the Tool Confidence Level concept that is used below.

A large variety of different command-line options may be specified to the `objcopy` command. Some of these options control or modify the generation of the resulting binaries. It is not feasible to test that all combinations of all options work correctly. The command-line options detailed below have been subject to extensive verification by BlackBerry QNX.

This verification provides a significant part of the justification for BlackBerry QNX qualifying the tool chain to TCL-3 under ISO 26262-8 and to T-3 under IEC 61508-3. Specifying any other options that affect the code generation must be considered carefully, as BlackBerry QNX has not verified the correct operation of additional options.

A.6.1 Allowed command-line options

A developer MAY specify the following options when copying object files in safety applications:

```
--strip-debug -R.ident
-V
```

Input	Restriction	Notes
file type	ELF binary, archive, shared library, static library	Target file cannot have a mixture of ELF architectures and/or non-ELF files
file size	100MB	Maximum size of file to be copied
file number	10,000	Maximum number of files in the target file
file format	5000 ELF sections (up to 500 different types)	Maximum number of ELF sections
file names	64 Characters	Maximum filename length (not including path)
file path	256 Characters	Maximum path length (Note:Windows limit is 292 for both file name and path)
link number	0-6 depth	Maximum number of symbolic link redirections (Not supported on Windows)
@file	not supported	

Table A.4: Restrictions on the objcopy utility

These are simply the options that BlackBerry QNX has explicitly verified.

Note that BlackBerry QNX recommends that options that affect the resultant binary, which are not listed above, be validated by the user according to either ISO 26262-8 or IEC 61508-3. BlackBerry QNX can provide services to assist users with the qualification of additional options. Further note that command-line options that do not affect the resultant binary are considered safe.

A.6.2 Restrictions

In order to limit the field of testability, Table A.4 lists the restrictions placed on the file which is being operated on by the `objcopy` command:

APPENDIX B

Checking the Integrity of Command-Line Options

Contents

B.1 Introduction

When the QOS is started, various command-line options to `procnto` may be set: see Appendix A.3 `procnto` command-line options. These options are held in RAM and are used by the kernel code at various decision points.

It may be desirable to verify at run time that the kernel has been configured correctly. Since the configuration data structures are constant[1], this need only be done once during system startup.

The QOS provides the `/proc/config` API that allows an application to examine the values of the kernel configuration data structures. This API is described in the QOS product documentation. An example of how it may be used is given in section Sample code below.

If the failure analysis of the application requires these values be checked at startup, the check should determine if the configured values are acceptable and, if not, the necessary action would be taken at the application level (e.g., moving the device to its Design Safe State).

[1] Note that the kernel stores its configuration data in data structures that, through duplication, are protected from external (e.g., cosmic ray resulting in a bit flip) or accidental (e.g., a software defect resulting in a write to random memory) corruption.

B.2 Sample code

The following is an example of a procedure you can use to validate the value of a particular kernel configuration parameter.

```
/* Confirm the nopreempt variable has value 2 */
int verifyConfiguration(void) {
    FILE *f;
    char data[128];

    /* Open configuration file  */
    if ((f = fopen ("/proc/config", "r")) == NULL) {
        printf("Failed to open 'config' file, errno %d (%s)\n",
                  errno, strerror (errno));
        return 0;
    }

    while (fgets (data, sizeof(data), f) != NULL) {
     if (strncmp(data, "nopreempt:", 10) == 0) {
         /* Found the nopreempt value */
         if (strncmp(data, "nopreempt:2", 11) == 0) {
             /* It has the correct value-- return success */
             fclose(f);
             return 1;
         } else {
             /* Wrong value-- return failure */
             fclose(f);
             return 0;
         }
     }
    }

    /* Didn't find it-- return failure. */
    fclose(f);
    return 0;
}
```

APPENDIX C

Reporting a Hardware Error

Contents

C.1 Introduction

Restriction 13 defines the circumstances under which a detected hardware error should be reported to the QOS. This appendix describes the mechanism for making that report.

C.2 Identification of the API

The header file that should be included is:

`sys/sysmgr.h`

The signature of the API call is:

`int sysmgr_reboot(void);`

C.3 Example

The following is an example of using the API call.

```
#include <sys/sysmgr.h>

int reportHWError(void)
    {
```

```
    sysmgr_reboot();

    /* note that we do not expect to reach the next statement */

    return EXIT_SUCCESS;
    }
```

APPENDIX D

Optional Training

Contents

D.1 Introduction

BlackBerry QNX provides discretionary training for technical personnel developing applications using the *QNX OS for Safety*. The following training courses are highly recommended for all designers and programmers working with the *QNX OS for Safety*.

For questions regarding these training modules and other training related to the QNX Neutrino RTOS, refer to `http://www.qnx.com/support/training/` or email `services@qnx.com`. If you have an active enhanced service plan, contact your account representative.

D.2 Course: Interpreting the QOS Safety Manual

Duration	$\frac{1}{2}$ day
Title	Interpreting the QOS Safety Manual (this document)
Prerequisites	Experience with application programming with the QNX Neutrino RTOS or completion of the Realtime Programming for the QNX Neutrino RTOS (course 107302) course
Module Summary	This module introduces the designer or programmer, assumed to be already familiar with programming for QNX Neutrino, with the necessary technical and procedural background to build an application that would be a candidate for IEC 61508, IEC 62304, or ISO 26262 certification.

This training module provides:

- An interpretation of this document within the context of an IEC 61508, IEC 62304, or ISO 26262 Software Safety Life Cycle
- Guidance on the use of the *QNX OS for Safety*
- Application software lifecycle requirements
- Notes on the design of *QNX OS for Safety* applications

D.3 Course: Developing a Dependable Application

Duration	$2\frac{1}{2}$ days
Title	Developing a Dependable Application
Prerequisites	Experience with design or implementation of an embedded system. This first day of this course has also been found to be of use for certification managers looking to deepen their understanding of the technology behind the development of a certifiable system.
Module Summary	This module introduces the designer or programmer, assumed to be already familiar with embedded programming, to: • the requirements of the relevant standard • a collection of tools for analyzing, designing, and programming an application that must meet precise requirements of availability and reliability The course involves practical exercises with the tools.
Tailoring	The course is focused, as appropriate, on industrial (IEC61508, etc.), medical (IEC62304/ISO14971), rail (EN5012x), or automotive (ISO26262) applications and the associated standards.

The course consists of the following sessions:

- Session 1: Background

  This session introduces the basic terminology of “dependability” (reliability/availability, faults/errors/failures, Heisenbugs, etc.) and addresses the statistical nature of software failures.

- Session 2: The Standards and Safety Cases

  This session focuses on the particular standard of interest (IEC 61508, IEC 62304, EN 5012x, or ISO 26262) and introduces the Safety Case for presenting the safety argument in a structured manner. Both Goal-Structuring Notation (GSN) and Bayesian-Belief Networks (BBNs) are covered as suitable notations for the Safety Case.

- Session 3: Budgeting System Dependability

  This session introduces and demonstrates some methods for estimating the dependability of systems, ranging from crude but quick techniques, such as Markov models, to more sophisticated techniques, such as Bayesian Fault Trees.

These skills are essential in the design process, and techniques such as Petri Net and Discrete-Event Simulation modelling are also extremely useful during implementation.

- Session 4: Preventing the Introduction of Faults

  If faults are not introduced into design or code, there is no need to remove them later. Amongst other things, this session covers various formal techniques for *proving* that a design is correct.

  It also covers the difficult but important area of demonstrating that the compiler, linker, and other toolchain elements are not introducing faults into the code during compilation.

- Session 5: Removing Faults before They Become Errors

  Testing is an essential part of demonstrating the dependability of a system but its importance is reducing in a world of multi-threaded code and multicore processors.

  This session introduces the new, risk-based software-testing standard (ISO 29119) and covers some advanced testing techniques including Combinatorial Testing. It also demonstrates some tools for module test generation and covers some deep static analysis tools (e.g., for symbolic execution) and fault injection for assessing the number of remaining Heisenbugs.

- Session 6: Preventing Errors from Becoming Failures

  If faults are introduced into design or code, they may cause errors to occur. This session describes and demonstrates some design and programming techniques for preventing errors turning into failure including recovery blocks, virtual synchrony, and data diversification (rather than replication). It also covers the crash-only approach to failure.

This course is accompanied by a 300-page textbook written by BlackBerry QNX.

APPENDIX E

Support Information

Contents

E.1 Support Information

BlackBerry QNX is committed to providing customers with exceptional support.

Support is available through a dedicated online portal, person-to-person help lines, community portal, knowledge base, and more. Here are some useful links:

Support portal — central hub for information, resources and issue management for all registered support plan customers: `http://www.qnx.com/account/login.html`

Support options — your choice of person-to-person help lines, dedicated technical resources, or a customized team of support professionals: `http://www.qnx.com/support/support.html`

Knowledge base — clearinghouse for tips, tricks, and technical articles: `http://www.qnx.com/support/knowledgebase.html`

E.2 New to QNX?

If you haven't had the opportunity to avail yourself of BlackBerry QNX's excellent support services in the past, here are some helpful starting points:

Accessing Online Technical Support — your first stop for help with setting up your myQNX account, registering your support plan, and using the portal: http://www.qnx.com/download/feature.html?programid=18239

Quickstart Guide: Five Steps to Developing a QNX Neutrino Program – an indispensable resource for help with installing and configuring the QNX Software Development Platform : http://www.qnx.com/developers/docs/7.1/#com.qnx.doc.qnxsdp.quickstart/topic/about.html

E.3 Reporting defects

As required by restriction 10, you are required to report defects that you find in the QOS so that BlackBerry QNX can assess the possible impact of the defect on the continued safe operation of the QOS.

Such defects should be reported through your support contact, whether standard or priority.

E.4 Exception for restrictions

Restrictions as they appear in this document exist to identify and/or suggest mitigations for residual risks identified during the BlackBerry QNX product safety lifecycle. The intent of a restriction may be deviated from, provided there is sufficient evidence for the product system safety case to justify that the associated risk(s) either do not apply in the context of use, or that they are mitigated by some other means to those proposed. BlackBerry Limited can work with any party to ensure there is sufficient understanding of the underlying risk and can provide technical opinion on a proposed alternative; however, permission from BlackBerry Limited to deviate a restriction is not required and will not be provided.

E.5 Direct contacts

For questions regarding this Safety Manual or *QNX OS for Safety* ISO 26262 or IEC 61508 certification, contact:

Louay Abdelkader, Product Manager
BlackBerry QNX
+1 519 8887465
labdelkader@blackberry.com

For questions regarding *QNX OS for Safety* technical support or related training, see http://www.qnx.com/support/index.html or http://www.qnx.com/support/training/ or contact services@qnx.com. If you have an enhanced service plan, contact your account representative.

APPENDIX F

Certified libslog2 Functions

The `libslog2` API is defined in the `<sys/slog2.h>` header file. The following `libslog2` functions are in certification scope and acceptable for use in a SIL/ASIL context:

- *slog2_register()**
- *slog2_reset()*
- *slog2f()*
- *vslog2f()*
- *slog2fa()*
- *vslog2fa()*
- *slog2c()*
- *slog2_get_verbosity()*
- *slog2_set_verbosity()**
- *slog2_set_default_buffer()*

Note that a star (*) beside a function name indicates that the function sends a message to `slogger2` and a timeout should be set when calling it.

The following functions are not in certification scope and are not acceptable for use in a SIL/ASIL context:

- *slog2_dump_logs_to_file()*
- *slog2_hash()*
- *slog2_obfuscate()*

In addition, the log parsing functions defined in `<slog2_parse.h>` are out of certification scope.

APPENDIX G

Clocks and Timers

Contents

The QOS provides number of clocks, which include `CLOCK_REALTIME`, `CLOCK_MONOTONIC`, and `CLOCK_SOFTIME`, for applications to fulfill their timing constraints. In the presence of timecc kernel module, the concept of time tracking with these clocks may be different. This appendix is an attempt to gather necessary details to avoid designer making false assumptions about the progress of time and the firing of timers armed based on the QOS provided clocks with and without timecc.

G.1 Without timecc

The QOS kernel uses the *timer_load()* kernel callout to program the hardware timer/counter chip to generate a periodic interrupt (aka system tick interrupt). If the hardware timer/counter chip does not auto-reset, the kernel uses the *timer_reload()* kernel callout to get periodic interrupt behavior. The QOS requires a periodic system tick interrupt to:

- update its clocks (e.g., `CLOCK_MONOTONIC`, and `CLOCK_REALTIME`),
- scan a list of timers to be fired,
- update per-process and per-thread CPU time clocks, and
- perform any necessary thread scheduling

G.1.1 Clocks Update at Every System Tick

By default, the QOS receives a periodic system tick interrupt at every 1 ms. Without timecc, the QOS clocks are updated as follows:

- `CLOCK_MONOTONIC` tracks the amount of time since the system booted. It begins from zero an arbitrary time during kernel initialization. For every system tick interrupt, the `CLOCK_MONOTONIC` is updated by exactly one *ClockPeriod()* per tick. However, due to the initial programming of the hardware timer for the system tick, the jitter in the system tick is reflected in `CLOCK_MONOTONIC` for a brief period (60 ticks). During this period of time at initialization, clock values don't move exactly one *ClockPeriod()* per tick.
- `CLOCK_REALTIME` is `CLOCK_MONOTONIC` plus offset. The offset can be calculated by subtracting the time passed since Jan 1, 1970, from the `CLOCK_MONOTONIC` accumulated since the system boot.
- `CLOCK_SOFTTIME` is same as `CLOCK_REALTIME`, but it can be used to specify the timer tolerance of an infinite value for a task not deemed as critical.

G.1.2 Timer Fire Times

The QOS provides two types of timers, i.e., relative and absolute. When a relative timer is armed based on `CLOCK_REALTIME` or `CLOCK_MONOTONIC`, the fire time = activation time + fire length + *ClockPeriod()*. The activation time is the current value of clock based on which timer was created, and the fire length is the relative fire time. The *ClockPeriod()* is added to ensure that timer fires late not early when checked against an independent wall clock.

After firing HRT or exiting from tickless, there is a period of time when the QOS kernel updates the time based on a diff between *ClockCycles()* samples instead of adding *ClockPeriod()* to `CLOCK_MONOTONIC`. This is because it has been observed that the hardware behind *timer_load()* may generate interrupts inconsistently for a period of time before settling down.

G.1.3 Timer Queues

Within the QOS, a timer is kept in either of two queues: the monotonic ("MON") queue, and the time-of-day ("TOD") queue.

The MON queue is for timers that were activated to fire at an absolute `CLOCK_MONOTONIC` time, or to fire at a relative time against any clock i.e., these are the timers that are unaffected by a change to the `CLOCK_REALTIME`. The TOD queue is for timers that were created using `CLOCK_REALTIME` and activated to fire at an absolute time.

In the system tick, the MON queue is scanned before the TOD queue. Therefore if the 50 timer-per-tick limit is hit while scanning the MON queue, no TOD queues will be fired.

This effectively gives timers in the MON queue "higher" priority than those in the TOD queue.

G.2 With timecc

timecc uses *ClockCycles()* as a consistent time basis. It foregoes the use of *timer_load()* and *timer_reload()* to program the timer hardware, since these callouts (that can support any hardware provided by startup) are relative-time based. Instead, timecc expects the hardware to be the lapic timer (in TSC-deadline mode) on x86_64 and the generic timer on aarch64. These pieces of hardware are known to be absolute-time based, with their 64-bit up counter being the same hardware as the one behind *ClockCycles()*. Since timecc takes a snapshot of *ClockCycles()* during its initialization (`boot_cc`) to create a reference point, the Nth tick in cycles can then be computed as `boot_cc` + N * *ClockPeriod()* (in cycles). That value can thus be fed directly in the hardware as an absolute value at which to trigger the clock interrupt.

G.2.1 Clocks Update at Every System Tick

At every system tick, the `CLOCK_MONOTONIC` is updated as:

$$T_{nanoseconds} = (ClockCycles()\ cycles\ -\ boot\ time) * \frac{1\ sec}{qtime->\ cycles\_per\_sec\ cycles} * \frac{10^9\ ns}{1\ sec}$$

Since the current time is based on the function of *ClockCycles()*, the difference between two samples of *ClockCycles()* may not be exact multiple of *ClockPeriod()*. This jitter in system tick, illustrated in Figure G.1, will be reflected in `CLOCK_MONOTONIC`, `CLOCK_REALTIME`, and `CLOCK_SOFTTIME`.

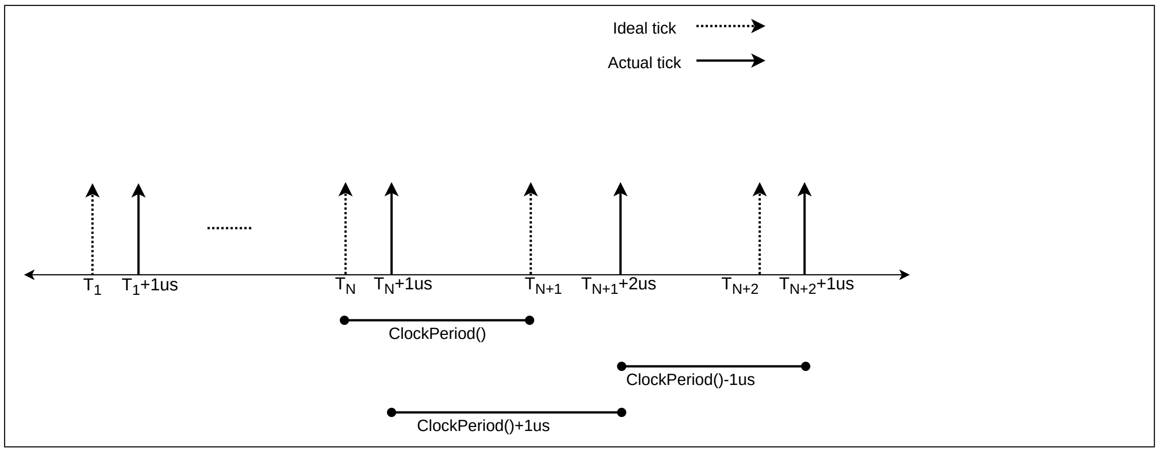

Figure G.1: Illustration of tick latency when an actual tick is delayed compared to ideal tick

G.2.2 Timer Fire Times

The timers in general behave similarly as without timecc. The timer fire times may vary depending on the jitter of the system tick (Figure G.1), and a timer fire may be delayed by an extra *ClockPeriod()*. For example, say the actual tick has latency of 3us and for some reason tick N was delayed by 5us, and `CLOCK_MONOTONIC` was updated to be T + 5us. At some point after the tick (but before the next tick) a single-shot timer is created based on `CLOCK_MONOTONIC`, and is activated to fire 10ms from now. The tick latency is adjusted back to 3us after the timer is armed. The timer's fire time is determined to be:

$$\begin{aligned} T_{fire} &= (T + 5us) + 10,000us + \textit{ClockPeriod()} \\ &= T + 11,005us \end{aligned}$$

However, when the tick N+11 arrives, it has a short latency, and the time is updated to be T + 11,0003 us. For this tick, the timer's fire time is still in the future, so it will not fire; it will be fired on the next tick.

APPENDIX H

QOS Resources and Constraint Mechanisms

Resource	Constraint Mechanism	Unconstrained Limit	Transient
private memory	`rlimit`	available memory	yes
shared memory	`rlimit`	available memory	no
side-channel connections	`rlimit`, *procmgr_ability()*, specific mechanisms implemented by resource manager	max 65,534 side-channel connections per process	yes
open files	`rlimit`, *procmgr_ability()*, POSIX permissions, *chroot()*/sandbox access, specific mechanisms implemented by resource manager	max 65,534 file descriptors per process	yes
process creation	`rlimit`, *procmgr_ability()*	4095 total processes in system	no
thread creation	`rlimit`	32,767 threads per process	yes
stack size	`rlimit`	available memory	yes

Resource	Constraint Mechanism	Unconstrained Limit	Transient
CPU time	`rlimit`, `aps`	each unconstrained thread might monopolize a single CPU core with respect to equal or lower priority threads	yes
sync object creation	available memory	available memory	yes
timer creation	`rlimit`	available memory	yes
registered events	`rlimit`	available memory	yes
pulses	N/A	available memory	yes

Table H.1: QOS Resources and Constraint Mechanisms

APPENDIX I

Summary of Restrictions and Recommendations

RST-0001. All users of the QOS SHALL be familiar with the latest version of release notes and bug reports (*QMS3263: Defect Notification*) shipped by BlackBerry QNX for the particular release being used.

RST-0002. All users of the QOS SHALL be familiar with any errata published for the hardware upon which their application will be run, and SHALL ensure that their application can function correctly in the presence of these errata.

RST-0003. The QOS SHALL be installed in accordance with the instructions in the *Installing and Using the QNX OS for Safety 2.2.2* guide.

RST-0004. A system that deploys the QOS SHALL NOT contain malicious application code.

RST-0005. The QOS SHALL be accessed from application code only through interfaces published in the BlackBerry QNX public documentation.

RST-0006. All developers creating applications to run on the QOS SHALL be aware of and SHALL follow the restrictions that the BlackBerry QNX library documentation (provided as part of the QNX SDP 7.1 installation) applies to the contexts in which different `libc` functions may be invoked.

RST-0007. All developers creating applications to run on the QOS SHALL be aware of the requirements and recommendations concerning command-line options, provided to components and utilities of the QOS, contained in appendix A on page 87. Developers SHALL follow the requirements contained in that appendix and SHOULD follow the recommendations contained in that appendix.

RST-0008. No kernel modules, other than those supplied by BlackBerry QNX, SHALL be added to the QOS.

RST-0009. No assumption SHALL be made about the timing of specific library calls other than where such timing guarantees are given explicitly in published BlackBerry QNX documentation.

RST-0010. If the user of the QOS detects that the QOS is not behaving in accordance with its published documentation, the user SHALL report that to BlackBerry QNX.

RST-0011. The system SHALL be engineered in such a way that exhaustion of any resource, in particular memory, does not occur.

RST-0012. The system within which the QOS is embedded SHALL detect the QOS moving to its Design Safe State and SHALL take the necessary action to preserve the safety of the system.

RST-0013. System self-checking mechanisms SHALL be put in place to detect internal hardware faults by utilizing hardware fault detection mechanisms and fault modes provided by hardware manufacturers. Based on application's criticality level, detected faults/errors SHALL be reported to QOS by utilizing hardware error reporting API in order to maintain the required level of integrity for the application.

RST-0014. The architecture of a safety application SHALL comply with any and all requirements of underlying components, including the hardware (e.g., processor or SoC) and BSP.

RST-0015. If QOS is deployed on a multiprocessor system, the system developer SHALL ensure that the hardware used to implement *ClockCycles()* is synchronized across all processor cores.

RST-0016. If the QOS is configured to use more than 8 CPU (application) cores, the system developer SHALL verify performance assumptions under the maximum application processing load.

RST-0017. The application SHALL NOT contain a continuously running, regular, high-frequency hardware interrupt unless the `-p` command-line option is used when starting the kernel.

RST-0019. If a system is deployed that depends upon a certain degree of clock accuracy to maintain a safety function, the system developer SHALL verify that the chosen clock is accurate enough to meet the requirements of the safety function in all expected environments and states, especially in the presence of Über-Authorities, if tickless is enabled, or High Resolution Timers are used.

RST-0020. If a system is deployed that makes extensive use of hardware or software timers, then the system developer SHALL verify the correct operation of their system during times of maximum timer fire frequency.

RST-0021. A system that includes hardware apart from the processor that has direct access to the system's memory (such as a bus master DMA device) SHALL employ hardware-level memory access constraints (such as an IOMMU) to ensure sufficient isolation between the separate applications and between the applications and the operating system.

RST-0023. An application SHALL NOT use atomic operations (including all mutex operations) in non-cacheable memory on AArch64-based systems.

RST-0024. On a multicore processor, the system tick interrupt SHALL be directed to core 0.

RST-0026. When using APS, if the design of a safety feature requires accuracy in the minimum partition budget for correct operation, then the safety feature SHALL be tested under high load conditions.

RST-0027. The application SHALL NOT rely on the correct detection of bankruptcy by the Adaptive Partitioning Scheduling (APS).

RST-0028. The APS scheduler SHALL NOT be used on multicore systems that provide different processing power on different cores (that is, asymmetric multicore processors).

RST-0029. A system SHALL NOT combine the use of the APS scheduling algorithm and power management where different cores can be configured to have different power levels.

RST-0030. When a channel is disconnected, an application SHALL NOT rely on the ordering of the delivery of the pulses.

RST-0033. An application SHALL NOT invoke the *dup2()* function while simultaneously creating other new file descriptors.

RST-0036. The system SHALL NOT execute the `mount_ifs` utility.

RST-0037. An application SHALL NOT modify opaque data structures created by the QOS.

RST-0038. The LD_BIND_NOW environment variable SHALL be set to prevent lazy function binding.

RST-0039. An application SHALL NOT create a robust mutex on a page of memory that was mapped into the application with the MAP_PHYS flag.

RST-0040. An application SHALL NOT use the *SyncMutexEvent()* function on a mutex created in a page of memory that was mapped into the application with the `MAP_PHYS` flag.

RST-0041. An application SHALL NOT use the -ad option to the `procnto` command line on AArch64 systems. Consult section A.3.2 `procnto` command-line options recommendations for details.

RST-0044. An application SHALL NOT set the *MALLOC_BAND_CONFIG_STR* environment variable.

RST-0045. An application SHALL NOT depend upon a thread that drops its own priority continuing to run if there are other ready threads of priority equal to the first thread's new priority.

RST-0046. Applications SHALL NOT depend upon an immediate timeout on a *MsgReceive()* call resetting the thread's APS partition.

RST-0047. Applications SHALL NOT depend upon the accuracy of scheduler time slicing when using the sporadic scheduling algorithm.

RST-0050. By default the clock period (the rate at which clock ticks arrive, in nanoseconds) will be 1 ms (1e+6 ns). If any other value is required, the value SHALL be established by calling *ClockPeriod()* with the desired value at boot time, as early as possible, and *ClockPeriod()* SHALL only be used thereafter for reading.

RST-0052. BSPs SHALL NOT include mini drivers, also known as IDA (Instant Device Activation).

RST-0053. Use of `libslog2` functions in a safety application SHALL be limited to those uses listed in Appendix F.

RST-0054. The `libslog2` library SHALL be used to register a buffer with the `slogger2` utility only during application startup.

RST-0055. The `libmod_qvm` kernel module SHALL be used if the system development also follows the Safety Manual of the QNX Hypervisor for Safety.

RST-0056. The `libfdt` library contains a QNX internal API and SHALL NOT be used by executables or libraries except those delivered by QNX.

RST-0057. Users of the `smmuman` utility to manage SMMU/IOMMU hardware MUST comply with the *SMMUMAN Safety Manual*.

RST-0058. Users of the `libm` library for mathematical functions MUST comply with the *Mathematics Library Safety Manual*.

RST-0059. When the QOS is running on top of a hypervisor, users of the QOS SHALL re-verify timing-related safety requirements in the *QOS Hazard and Risk Analysis* that are deemed relevant to the intended safety function.

RST-0060. When the QOS is running on top of a hypervisor, users of the QOS SHALL comply with the safety manual of the hypervisor.

RST-0061. A safety application SHALL NOT be built with both the `-g` and `-flto` command-line options provided simultaneously to the compiler and/or linker.

RST-0062. A safety application SHALL NOT be built with both the `-O` *n* and `-ftree-vectorize` command-line options provided simultaneously to the compiler and/or linker.

RST-0063. The `qcc/q++` compiler SHALL be used to compile safety critical applications.

RST-0065. The reboot kernel callout provided by the BSP SHALL reset the system in a timely manner.

RST-0066. The `smmuman-safety` application MUST be spawned with the `POSIX_SPAWN_CRITICAL` attribute.

RST-0067. The `server-monitor` application SHALL be spawned with the `POSIX_SPAWN_CRITICAL` attribute.

RST-0068. A process SHALL NOT *mmap_peer()* memory into the address space of another process with a higher SIL/ASIL.

RST-0069. A process SHALL NOT *mmap_peer()* into the address space of another process with a lower SIL/ASIL if the corruption of that memory could lead to incorrect behavior of another process with a higher SIL/ASIL.

RST-0070. A system SHALL NOT terminate a critical process during normal shutdown.

RST-0071. The termination of any non-critical processes SHALL NOT cause any critical processes to terminate.

RST-0072. A system SHALL NOT have any processes with the `PROCMGR_AID_TRACE` ability in normal mode of operation.

RST-0073. The `server-monitor` process SHALL run at a higher priority than all of the processes it is monitoring.

RST-0074. To reduce the potential impacts on QOS performance, those Über-Authorities that are not critical to the correct and safe operation of the system SHALL be removed from the system.

RST-0075. When an Über-Authority takes control, it SHALL save the programmer-visible state of all processor cores to be restored before releasing control.

RST-0076. A system designer SHALL fill in the duration from Restriction 77 into the system page *qtime* field *timer_prog_time* if either:

- the system tick timer is being modified at any time (e.g., HRTs, changes to tickless operation mode, changing *ClockPeriod()*), or
- any Über-Authorities are present.

If the startup supports the `-f` option, it can be used instead of writing the syspage *qtime*. More details can be found in *QMS3310: Startup Module High-Level Design*.

RST-0077. A system designer SHALL estimate the period of time required to program the system tick hardware timer for their hardware.

RST-0078. A threshold value for the maximum difference between *ClockCycles()* on all cores allocated to QOS SHALL be specified in the presence of Über-Authorities.

RST-0079. The sampling with *ClockCycles()* SHALL be based on a hardware-based source.

RST-0080. The hardware upon which *ClockCycles()* is based SHALL be synchronized with the time reference of the QOS execution environment, which can be a physical processor or an Über-Authority, such as a hypervisor.

RST-0081. The hardware used to implement *ClockCycles()* SHALL NOT roll over while QOS is in use (i.e., between power up and power off).

RST-0082. The frequency of the hardware source for *ClockCycles()* SHALL not vary with the power state of the CPUs and system, for the duration of use from power up to shut down.

RST-0083. The *ClockCycles()* frequency SHALL be specified in startup by setting the *qtime* field *cycles_per_sec*.

RST-0084. When porting an old code base to the new QOS release, all uses of the new implementation of *nanospin_ns()* SHALL be examined to confirm that the values given to *nanospin_ns()* continue to meet timing needs.

RST-0085. System designers SHALL perform a safety analysis of QOS's execution environment and consider the following when performing the analysis:

- hardware errata
- known issues with firmware and firmware configurations, and
- if QOS is running as a guest, known issues with the hypervisor within which it is running

The user SHALL ensure that their application can function correctly in the presence of errata or known issues.

RST-0086. Any shared or static library linked or loaded by a certified application running on QOS SHALL be certified to the same SIL/ASIL level as the application.

RST-0087. System integrators SHALL implement any necessary mitigations against random hardware faults.

RST-0089. System integrators SHALL employ a strategy at the system level to ensure that the program loader and QNX Startup Module have correctly initialized the system.

RST-0090. System integrators SHALL certify kernel callouts in context with the kernel.

RST-0092. In tickless mode, an interrupt service routine (ISR) SHALL NOT make use of *SYSPAGE_ENTRY(qtime)->nsec*.

RST-0093. A process using *mmap_peer()* SHALL have its `PROCMGR_AID_MEM_PEER` ability restricted (i.e., specifying a subrange of peer processes for the ability) such that it may not *mmap_peer()* into a process with a higher SIL/ASIL.

RST-0094. The run state of processor cores SHALL NOT be changed. The functions *sysmgr_runstate()*, *sysmgr_runstate_dynamic()*, and *sysmgr_runstate-burst()* that are used for changing a core's run state SHALL NOT be used in certified applications.

RST-0095. The function *InterruptHookIdle2()*, which is used to attach an `_NTO_HOOK_IDLE` synthetic interrupt handler when a processor becomes idle, SHALL NOT be used in certified applications.

RST-0096. The *SchedCtl()* command `SCHED_CONFIGURE` SHALL NOT be used in certified applications.

RST-0097. The *LIBC_FATAL_STDERR_* environment variable SHALL be set to 1 in certified applications.

RST-0098. If the thread's stack is specified using *pthread_attr_setstack()*, *pthread_attr_setstackaddr()*, or *ThreadCreate()*, then the thread stack SHALL have a guard page of at least `_SC_PAGE_SIZE` bytes immediately before the lowest stack address.

RST-0099. The guard page size for a QOS-provided thread stack SHALL NOT be set to 0, i.e., the guard page size SHALL be either left at the default size, or set to a value provided that is greater than the default size.

RST-0100. When APS is used, APS security SHALL be enabled using the *SchedCtl()* command SCHED_APS_ADD_SECURITY, and SHALL be enabled immediately using the highest security setting appropriate for the system.

RST-0101. Sigevents of type `SIGEV_THREAD` SHALL NOT be used.

RST-0102. The binary `secpol.bin` SHALL be stored as read-only on the IFS.

RST-0103. The `secpolgenerate` utility SHALL never run in a production environment.

RST-0104. When configuring a security policy file, only those `libsecpol` functions described in the QOS documentation SHALL be used.

RST-0105. When configuring security policy files, the following functions SHALL NOT be used:

- *secpol_find_blob()*
- *secpol_find_custom_blob()*
- *secpol_reset()*
- *secpol_find_entry()*
- *secpol_crc32()*

RST-0106. The `procnto` option `-bl` SHALL be used to avoid errors caused by calls to *procmgr_ability()* when a process attempts to configure locked abilities.

RST-0107. A client SHALL NOT register an event for all servers (i.e., pass a *coid* of -1 to *MsgRegisterEvent()*) unless the client is prepared to receive that event from all processes to which it establishes a connection.

RST-0108. Users of the `libcxx` Certified C++ Library MUST comply with the *libcxx Safety Manual*.

RST-0109. The user SHALL NOT modify any of the following:

- any header files associated with any certified libraries (e.g., `libc`) distributed by BlackBerry QNX, either as part of QOS or as a standalone product.
- preprocessor macros defined by these header files
- preprocessor macros predefined by the compiler (e.g., __QNX__)
- all objects associated (e.g., __`func`__) with reserved identifiers (ISO/IEC 9899:2011 7.1.3).

RST-0110. All safety critical applications SHALL be compiled with the `qcc/q++` compiler using only the options listed in A.1.1.

RST-0111. No compiler builtins SHALL be explicitly called from any application that is compiled with the `qcc/q++` compiler.

RST-0112. A dumper process SHALL register the path /**proc**/**dumper** only after all preparations required to receive dumper messages are complete.

RST-0113. A dumper process SHALL respond to all resource manager messages received on the path /**proc**/**dumper**.

RST-0114. A dumper process SHALL return either of the following return values:

- -1 if a core dump could not, or was not, successfully created, otherwise
- the size, in bytes, of the dumper message.

RST-0115. A dumper process SHALL register the path /**proc**/**dumper** as a file of type `_FTYPE_DUMPER`.

RST-0116. Any process that registers the path /**proc**/**dumper** with file type `_FTYPE_DUMPER` SHALL be safety-certified to the highest safety integrity level allocated to the software system.

RST-0117. Any process in a deployed system given access to the directory /**proc**/***pid*** and all files beneath SHALL be safety-certified to the highest safety integrity level allocated to the software system.

RST-0118. Any access to /**proc**/**dumper** SHALL be restricted to only those processes which require access to it.

RST-0119. Wide characters SHALL NOT be used in the code.

RST-0121. All projects using QOS 2.2.2 SHALL compile or recompile all binaries and their composite object files using the latest version of QOS 2.2.2 toolchain.

RST-0122. Only QNX qualified tools SHALL be used to modify compiled binaries.

RST-0123. The startup binary SHALL ensure that the system page `qtime`, field *rr_interval_mul* value is set to 0.

RST-0124. A maximum of 8 APS partitions SHALL be configured on QOS.

RST-0125. If a channel is created with a pulse pool, and the flag `_NTO_CHO_CUSTOM_EVENT` is passed to request notification of failure to deliver a pulse, then:

- a sigevent of type `SIGEV_SEM` SHALL be used, and
- the rearm threshold SHALL be set to a value greater than the number of pulses in the pulse pool.

RST-0127. An application thread that acquires I/O privileges with *ThreadCtl()* SHALL acquire the lowest I/O privilege level necessary.

RST-0128. The *ThreadCtl()* command flag `_NTO_TCTL_IO_LEVEL_INHERIT` SHALL be set only when a thread with I/O privileges must create another thread requiring I/O privileges.

RST-0129. If the custom kernel call callout is NOT NULL, then a system integrator SHALL perform a safety analysis of the implementation of the custom kernel call callout to the same ASIL/SIL level as the QOS.

RST-0130. If the custom kernel call callout is NOT NULL, then a system integrator SHALL perform a security analysis of the implementation of the custom kernel call callout.

RST-0131. If pointer authentication functionality is used in a system, then the system integrator SHALL take appropriate measures to ensure the functionality performs correctly, and

RST-0132. If FEAT_SSBS functionality is used in a system, then the system integrator SHALL take appropriate measures to ensure the functionality performs correctly.

RST-0133. Applications SHALL NOT depend upon the accuracy of representation when converting between string and real number types.

RST-0134. If timecc is used in a production QOS system, the safety variant of timecc, i.e., `libmod_timecc-safety.so`, SHALL be used.

RST-0136. A process SHALL NOT attach an interrupt handler to the system tick interrupt, i.e., `SYSPAGE_ENTRY(qtime)->intr`.

RST-0138. Since no safety analysis of pointer authentication functionality has been performed, the system integrator SHALL ensure that the system using pointer authentication functionality satisfies the requirements of the system's safety level.

RST-0139. Since no safety analysis of FEAT_SSBS functionality has been performed, the system integrator SHALL ensure that the system using FEAT_SSBS functionality satisfies the requirements of the system's safety level.

RST-0140. The QOS configuration SHALL be using the safety variant of the kernel and all (if any) kernel modules.

RST-0141. Unless a design explicitly tolerates dropped pulses, a designer SHALL ensure that if a notification of a dropped pulse is received, then:

- a notification of a dropped pulse SHALL cause the channel's process to enter its DSS, and
- that a sigevent of type `SIGEV_SEM` SHALL be used in lieu of `SIGKILL` only to perform a small amount of cleanup before entering the DSS.

REC-0001. The assumptions made about inter-process communications within the QOS by design tools used to generate and validate high- and low-level designs SHOULD be identified and validated to ensure that they correctly reflect the actual operation of the QOS.

REC-0002. Before building a target image, the user SHOULD check the integrity of all of the components of the QOS.

REC-0003. The integrity of certified binaries SHOULD be checked at runtime.

REC-0004. The recommended command line options in Section A.3.2 SHOULD be used to start `procnto`.

REC-0005. The system designer SHOULD make use of the flexibility provided by the QOS for placing access restrictions on servers (resource managers).

REC-0006. Applications SHOULD allocate resources required to implement safety functions early in their life cycle. In particular, large memory allocations, allocations of physically contiguous memory and any other resource on which the operating system does not impose constraints SHOULD be completed at application startup.

REC-0008. A system SHOULD NOT make use of dynamic power management.

REC-0009. If dynamic power management is used, the system designer SHOULD ensure that the minimum requirements of all safety functions are met by the lowest power state that the system may enter.

REC-0010. Applications SHOULD NOT execute with **root** privileges unless those privileges are required.

REC-0011. Applications SHOULD release **root** privileges as soon as those privileges are no longer required.

REC-0012. System developers SHOULD ensure that applications are granted only those permissions necessary to implement their functionality.

REC-0013. The APS scheduler can be configured dynamically, but this is not recommended. The APS configuration SHOULD be established at system startup. Once a system has begun providing a safety function, APS SHOULD NOT be reconfigured.

REC-0014. The system designer SHOULD avoid the use of APS critical budgets.

REC-0015. The system designer SHOULD ensure that the scheduling design of the system is kept simple enough that it is possible to model the scheduling behavior of the system with sufficient accuracy to meet design requirements.

REC-0016. The system designer SHOULD characterize the jitter of the system tick ISR under low and heavy load on any system to be developed.

REC-0017. A watchdog timer or other mechanism SHOULD be implemented at the system level in order to detect when the QOS has moved to its design safe state (DSS) or has become unresponsive in the case of silent failure.

REC-0018. An application running on the QOS SHOULD confirm at application startup that the correct kernel is actually executing.

REC-0019. The version of the QOS binaries SHOULD be checked at runtime.

REC-0020. Threads that are subjected to different priority limits SHOULD NOT share resources that provide priority inheritance (e.g., mutexes).

REC-0021. The QOS SHOULD be configured to ensure mutexes that are shared between processes are safe.

REC-0023. Any resource manager invoked from another resource manager SHOULD be engineered in such a way that it always has at least one thread RECEIVE-blocked.

REC-0024. The integrity of the stored values of the `procnto` command-line options MAY be checked at runtime.

REC-0026. Applications SHOULD free stack memory when manually allocating thread stacks.

REC-0028. An application SHOULD NOT make use of emulated floating point (as distinct from hardware floating point) for computations that are safety-critical.

REC-0029. The system programmer SHOULD NOT use POSIX *fork()* in a multithreaded process.

REC-0031. A developer SHOULD NOT use inline assembly in the development of an application for a QOS system.

REC-0032. A developer SHOULD NOT use thread cancellation in the development of a safe application.

REC-0033. The unblock pulse timeout value set for the server monitor SHOULD account for the expected worst case response time among all monitored servers and allow sufficient slack to account for IPC overhead.

REC-0034. All high-priority or safety-critical clients requiring the server monitor service SHOULD be started after the server monitor.

REC-0035. Another process SHOULD verify that the server monitor is running correctly using the `_PROC_SERVER_MONITOR_ALIVE` message to `procmgr` during system initialization.

REC-0036. The system SHOULD be restarted or shut down if the return status of a `_PROC_SERVER_MONITOR_ALIVE` message is not `EOK`.

REC-0037. The `slog2info` utility SHOULD NOT be used to read buffers during normal mode of operation.

REC-0038. Clients SHOULD set a timeout before calling *slog2_register()* and *slog2_set_verbosity()*.

REC-0039. The `slogger2` process SHOULD be registered with the server monitor with the SIGKILL signal.

REC-0040. System integrators SHOULD spawn a process with the `SPAWN_CRITICAL` attribute if the system cannot operate safely after the process terminates or the process cannot be safely restarted.

REC-0041. A maximum number of retries to write an event to the buffer SHOULD be set for each log buffer.

REC-0043. A developer SHOULD NOT create a client connection with a server enabling the delivery of unregistered events. A client can allow the delivery of unregistered events by setting the flag `_NTO_COF_UNREG_EVENTS` when it calls *ConnectAttach()* to connect to server.

REC-0045. The current execution state of a processor core (e.g., halted, in privileged state, in user state) SHOULD be taken into account before Über-Authorities attempt to gain exclusive access and control of the QOS.

REC-0046. If the removal of all Über-Authorities is not possible, then the potential impacts of remaining Über-Authorities SHOULD be taken into consideration.

REC-0047. When setting timeout thresholds for kernel callouts or client-server communication, a designer SHOULD take Über-Authorities into account.

REC-0048. If any process maps to specific physical address ranges, then those ranges SHOULD be checked for unintended overlap with address ranges assigned to other processes.

REC-0049. Typed memories (created by calling *posix_typed_mem_open()*) SHOULD be used instead of hard-coded physical addresses, where possible.

REC-0050. Processes mapping memory with `MAP_FIXED` and having the `PROCMGR_AID_MAP_FIXED` ability SHOULD NOT create mappings at virtual address 0.

REC-0051. The use of fortified system functions SHOULD be enabled at level 2, which is the highest fortification level supported.

REC-0052. The integrity of the QOS Image Filesystem (IFS) SHOULD be checked when loading the system (e.g., Secure Boot)

REC-0053. The performance of LSE (Large System Extension) atomics SHOULD be compared with LLSC (Load-Link/Store-Conditional) atomics on any ARMv8.1-A system to be developed.

REC-0054. Immediately after APS has been configured, it SHOULD be locked using SCHED_APS_SEC_PARTITIONS_LOCKED to prevent any further unauthorized changes.

REC-0055. The system page SHOULD be periodically checked for corruption.

REC-0056. The policy file generated by `secpolgenerate` SHOULD be reviewed and examined to ensure that the privileges specified for processes are assigned as expected.

REC-0057. When developing a security policy, ranges for process privileges SHOULD be specified where possible.

REC-0058. In the security policy, security types that are permitted to spawn SHOULD be configured to change type automatically with limited privileges on spawn.

REC-0059. If a security policy is enforced, resource managers SHOULD drop their privileges after initialization.

REC-0060. If an Über-Authority is absolutely necessary, then it SHOULD be implemented as a cooperative Über-Authority, and NOT as a non-cooperative Über-Authority, if possible.

REC-0061. The system developer SHOULD generate and examine the map files for all the applications to ensure that the binary files are using the correct versions of the QNX Software Development Platform (SDP), and all the dependent libraries.

REC-0062. *InterruptAttachEvent()* SHOULD be used instead of Interrupt Service Routines (ISRs).

REC-0063. Non-Maskable Interrupts (NMIs) SHOULD NOT be used.

REC-0064. A system SHOULD use POSIX signals only either:

- synchronously on the receiver end, i.e. masked at all times and only received by *sigwaitinfo()* or equivalent, or:
- when used to terminate processes.

REC-0065. Non-safety (i.e. no SIL/ASIL) qualified resource managers SHOULD use a static configuration of the smmuman with a typed memory setup.

REC-0066. A non-safety certified resource manager SHOULD NOT use the *smmuman* API dynamically at runtime.

REC-0067. Any channel created by a process SHOULD be created with a pulse pool of size dependent upon:

- the consumption rate of service thread(s) receiving on the channel,
- the number of clients, and
- the characteristics of communication between client and server.

REC-0068. An application thread that requires I/O privileges to perform work SHOULD only acquire I/O privileges when necessary. After performing work that requires I/O privileges, the thread SHOULD relinquish or reduce I/O privileges either by adjusting them with *ThreadCtl()*, or terminating.

REC-0069. The *ThreadCtl()* SHOULD be used with command flag `_NTO_TCTL_IO_LEVEL` instead of `_NTO_TCTL_IO` and `_NTO_TCTL_IO_PRIV`.

REC-0070. A system integrator SHOULD ensure the following, if applicable:

- The functionality of the custom kernel call is restricted based on a process's type id, and
- If the process's type id does not permit execution of the custom kernel call, an error code of the implementer's choosing must be returned.

REC-0071. The kernel module timecc SHOULD be used.

REC-0073. A system integrator SHALL characterize the drift of `CLOCK_MONOTONIC` against the external environment's time basis over a period of use and under conditions representative of expected use.

REC-0074. The system tick interrupt on core 0 SHOULD NOT be delayed for more than half of the tick system interval.

REC-0075. In environments where there is significant preemption, and there are processes with a significant number of threads, channels, or connections, the system designer SHOULD perform the timing analysis to analyze whether the thread progression issue is applicable on their target platform.