Xen Documentation

Release 4.18-unstable

The Xen development community

Aug 09, 2023

Contents

1 User documentation **3**

1.1 Admin Guide 3

2 Guest documentation **7**

2.1 Guest documentation 7

3 Hypervisor developer documentation **9**

3.1 Hypervisor documentation 9

4 MISRA C coding guidelines **13**

4.1 MISRA C rules for Xen 13

5 Miscellanea **21**

5.1 Glossary 21

Index **23**

Note: Xen's Sphinx/RST documentation is a work in progress. The existing documentation can be found at https://xenbits.xen.org/docs/

Xen is an open source, bare metal hypervisor. It runs as the most privileged piece of software on the system, and shares the resources of the hardware between virtual machines. See *Introduction* for an introduction to a Xen system.

CHAPTER 1

User documentation

This is documentation for an administrator of a Xen system. It is intended for someone who is not necesserily a developer, has installed Xen from their preferred distribution, and is attempting to run virtual machines and configure the system.

1.1 Admin Guide

1.1.1 Introduction

Xen is an open source, bare metal hypervisor. It runs as the most privileged piece of software, and shares the resources of the hardware between virtual machines.

In Xen terminology, there are *domains*, commonly abbreviated to dom, which are identified by their numeric *domid*.

When Xen boots, dom0 is automatically started as well. Dom0 is a virtual machine which, by default, is granted full permissions[1]. A typical setup might be:

Dom0 takes the role of *control domain*, responsible for creating and managing other virtual machines, and the role of *hardware domain*, responsible for hardware and marshalling guest I/O.

Xen is deliberately minimal, and has no device drivers[2]. Xen manages RAM, schedules virtual CPUs on the available physical CPUs, and marshals interrupts.

Xen also provides a hypercall interface to guests, including event channels (virtual interrupts), grant tables (shared memory), on which a lot of higher level functionality is built.

[1] A common misconception with Xen's architecture is that dom0 is somehow different to other guests. The choice of id 0 is not an accident, and follows in UNIX heritage.

[2] This definition might be fuzzy. Xen can talk to common serial UARTs, and knows how to drive various CPU internal devices such as IOMMUs, but has no knowledge of network cards, disks, etc. All of that is the hardware domains responsibility.

1.1.2 Microcode Loading

Like many other pieces of hardware, CPUs themselves have errata which are discovered after shipping, and need to be addressed in the field. Microcode can be considered as firmware for the processor, and updates are published as needed by the CPU vendors.

Microcode is included as part of the system firmware by an OEM, and a system firmware update is the preferred way of obtaining updated microcode. However, this is often not the most expedient way to get updates, so Xen supports loading microcode itself.

Distros typically package microcode updates for users, and may provide hooks to cause microcode to be automatically loaded at boot time. Consult your dom0 distro guidance for microcode loading.

Microcode can make almost arbitrary changes to the processor, including to software visible features. This includes removing features (e.g. the Haswell TSX errata which necessitated disabling the feature entirely), or the addition of brand new features (e.g. the Spectre v2 controls to work around speculative execution vulnerabilities).

Boot time microcode loading

Where possible, microcode should be loaded at boot time. This allows the CPU to be updated to its eventual configuration before Xen starts making setup decisions based on the visible features.

Xen will report during boot if it performed a microcode update:

```
[root@host ~]# xl dmesg | grep microcode
(XEN) microcode: CPU0 updated from revision 0x1a to 0x25, date = 2018-04-02
(XEN) microcode: CPU2 updated from revision 0x1a to 0x25, date = 2018-04-02
(XEN) microcode: CPU4 updated from revision 0x1a to 0x25, date = 2018-04-02
(XEN) microcode: CPU6 updated from revision 0x1a to 0x25, date = 2018-04-02
```

The exact details printed are system and microcode specific. After boot, the current microcode version can obtained from with dom0:

```
[root@host ~]# head /proc/cpuinfo
processor    : 0
vendor_id    : GenuineIntel
cpu family   : 6
model        : 60
model name   : Intel(R) Xeon(R) CPU E3-1240 v3 @ 3.40GHz
stepping     : 3
microcode    : 0x25
cpu MHz      : 3392.148
cache size   : 8192 KB
physical id  : 0
```

Loading microcode from a single file

Xen handles microcode blobs in the binary form shipped by vendors, which is also the format which the processor accepts. This format contains header information which Xen and various userspace tools can use to identify the correct blob for a specific CPU.

Tools such as Dracut will identify the correct blob for the current CPU, which will be a few kilobytes, for minimal overhead during boot.

Additionally, Xen is capable of handling a number of blobs concatenated together, and will locate the appropriate blob based on the header information.

This option is less efficient during boot, but may be preferred in situations where the exact CPU details aren't known ahead of booting (e.g. install media).

The file containing the blob(s) needs to be accessible to Xen as early as possible.

- For multiboot/multiboot2 boots, this is achieved by loading the file as a multiboot module. The `ucode=$num` command line option can be used to identify which multiboot module contains the microcode, including negative indexing to count from the end.
- For EFI boots, there isn't really a concept of modules. A microcode file can be specified in the EFI configuration file with `ucode=$file`. Use of this mechanism will override any `ucode=` settings on the command line.

Loading microcode from a Linux initrd

For systems using a Linux based dom0, it usually suffices to install the appropriate distro package, and add `ucode=scan` to Xen's command line.

Xen is compatible with the Linux initrd microcode protocol. The initrd is expected to be generated with an uncompressed CPIO archive at the beginning which contains contains one of these two files:

```
kernel/x86/microcode/GenuineIntel.bin
kernel/x86/microcode/AuthenticAMD.bin
```

The `ucode=scan` command line option will cause Xen to search through all modules to find any CPIO archives, and search the archive for the applicable file. Xen will stop searching at the first match.

Runtime microcode loading

Warning: If at all possible, microcode updates should be done by firmware updates, or at boot time. Not all microcode updates (or parts thereof) can be applied at runtime.

Given the proprietary nature of microcode, we are unable to make any claim that runtime microcode loading is risk-free. Any runtime microcode loading needs adequate testing on a development instance before being rolled out to production systems.

The `xen-ucode` utility can be used to initiate a runtime microcode load:

```
[root@host ~]# xen-ucode
xen-ucode: Xen microcode updating tool
Usage: xen-ucode <microcode blob>
[root@host ~]#
```

The details of microcode blobs (if even packaged to begin with) are specific to the dom0 distribution. Consult your dom0 OS documentation for details. One example with a Linux dom0 on a Haswell system might look like:

```
[root@host ~]# xen-ucode /lib/firmware/intel-ucode/06-3c-03
[root@host ~]#
```

It will pass the blob to Xen, which will check to see whether the blob is correct for the processor, and newer than the running microcode.

If these checks pass, the entire system will be rendezvoused and an update will be initiated on all CPUs in parallel. As with boot time loading, diagnostics will be put out onto the console:

```
[root@host ~]# xl dmesg | grep microcode
(XEN) microcode: CPU0 updated from revision 0x1a to 0x25, date = 2018-04-02
(XEN) microcode: CPU2 updated from revision 0x1a to 0x25, date = 2018-04-02
(XEN) microcode: CPU4 updated from revision 0x1a to 0x25, date = 2018-04-02
(XEN) microcode: CPU6 updated from revision 0x1a to 0x25, date = 2018-04-02
(XEN) 4 cores are to update their microcode
(XEN) microcode: CPU0 updated from revision 0x25 to 0x27, date = 2019-02-26
(XEN) microcode: CPU4 updated from revision 0x25 to 0x27, date = 2019-02-26
(XEN) microcode: CPU2 updated from revision 0x25 to 0x27, date = 2019-02-26
(XEN) microcode: CPU6 updated from revision 0x25 to 0x27, date = 2019-02-26
```

CHAPTER 2

Guest documentation

This documentation concerns the APIs and ABIs available to guests. It is intended for OS developers trying to use a Xen feature, and for Xen developers to avoid breaking things.

2.1 Guest documentation

2.1.1 x86

Hypercall ABI

Hypercalls are system calls to Xen. Two modes of guest operation are supported, and up to 6 individual parameters are supported.

Hypercalls may only be issued by kernel-level software[1].

Registers

The registers used for hypercalls depends on the operating mode of the guest.

ABI	Hypercall Index	Parameters (1 - 6)	Result
64bit	RAX	RDI RSI RDX R10 R8 R9	RAX
32bit	EAX	EBX ECX EDX ESI EDI EBP	EAX

32 and 64bit PV guests have an ABI fixed by their guest type. The ABI for an HVM guest depends on whether the vCPU is operating in a 64bit segment or not[2].

[1] For HVM guests, `HVMOP_guest_request_vm_event` may be configured to be usable from userspace, but this behaviour is not default.

[2] While it is possible to use compatibility mode segments in a 64bit kernel, hypercalls issues from such a mode will be interpreted with the 32bit ABI. Such a setup is not expected in production scenarios.

Parameters

Different hypercalls take a different number of parameters. Each hypercall potentially clobbers each of its parameter registers; a guest may not rely on the parameter registers staying the same. A debug build of Xen checks this by deliberately poisoning the parameter registers before returning back to the guest.

Mode transfer

The exact sequence of instructions required to issue a hypercall differs between virtualisation mode and hardware vendor.

Guest	Transfer instruction
32bit PV	INT 0x82
64bit PV	SYSCALL
Intel HVM	VMCALL
AMD HVM	VMMCALL

To abstract away the details, Xen implements an interface known as the Hypercall Page. This allows a guest to make a hypercall without needing to perform mode-specific or vendor-specific setup.

Hypercall Page

The hypercall page is a page of guest RAM into which Xen will write suitable transfer stubs.

Creating a hypercall page is an isolated operation from Xen's point of view. It is the guests responsibility to ensure that the hypercall page, once written by Xen, is mapped with executable permissions so it may be used. Multiple hypercall pages may be created by the guest, if it wishes.

The stubs are arranged by hypercall index, and start on 32-byte boundaries. To invoke a specific hypercall, `call` the relevant stub[3]:

```
call hypercall_page + index * 32
```

There result is an ABI which is invariant of the exact operating mode or hardware vendor. This is intended to simplify guest kernel interfaces by abstracting away the details of how it is currently running.

Creating Hypercall Pages

Guests which are started using the PV boot protocol may set set `XEN_ELFNOTE_HYPERCALL_PAGE` to have the nominated page written as a hypercall page during construction. This mechanism is common for PV guests, and allows hypercalls to be issued with no additional setup.

Any guest can locate the Xen CPUID leaves and read the *hypercall transfer page* information, which specifies an MSR that can be used to create additional hypercall pages. When a guest physical address is written to the MSR, Xen writes a hypercall page into the nominated guest page. This mechanism is common for HVM guests which are typically started via legacy means.

[3] `HYPERCALL_iret` is special. It is only implemented for PV guests and takes all its parameters on the stack. This stub should be `jmp`'d to, rather than `call`'d. HVM guests have this stub implemented as `ud2a` to prevent accidental use.

CHAPTER 3

Hypervisor developer documentation

This is documentation for a hypervisor developer. It is intended for someone who is building Xen from source, and is running the new hypervisor in some kind of development environment.

3.1 Hypervisor documentation

3.1.1 Code Coverage

Xen can be compiled with coverage support. When configured, Xen will record the coverage of its own basic blocks. Being a piece of system software rather than a userspace, it can't automatically write coverage out to the filesystem, so some extra steps are required to collect and process the data.

Warning: ARM doesn't currently boot when the final binary exceeds 2MB in size, and the coverage build tends to exceed this limit.

Compiling Xen

Coverage support is dependent on the compiler and toolchain used. As Xen isn't a userspace application, it can't use the compiler supplied library, and instead has to provide some parts of the implementation itself.

For x86, coverage support was introduced with GCC 3.4 or later, and Clang 3.9 or later, and Xen is compatible with these. However, the compiler internal formats do change occasionally, and this may involve adjustments to Xen. While we do our best to keep up with these changes, Xen may not be compatible with bleeding edge compilers.

To build with coverage support, enable `CONFIG_COVERAGE` in Kconfig. The build system will automatically select the appropriate format based on the compiler in use.

The resulting binary will record its own coverage while running.

Accessing the raw coverage data

The `SYSCTL_coverage_op` hypercall is used to interact with the coverage data. A dom0 userspace helper, `xenconv` is provided as well, which thinly wraps this hypercall.

The `read` subcommand can be used to obtain the raw coverage data:

```
[root@host ~]# xencov read > coverage.dat
```

This is toolchain-specific data and needs to be fed back to the appropriate programs to post-process.

Alternatively, the `reset` subcommand can be used reset all counters back to 0:

```
[root@host ~]# xencov reset
```

GCC coverage

A build using GCC's coverage will result in `*.gcno` artefact for every object file. The raw coverage data needs splitting to form the matching `*.gcda` files.

An example of how to view the data is as follows. It uses `lcov` which is a graphical frontend to `gcov`.

- Obtain the raw coverage data from the test host, and pull it back to the build working tree.
- Use `xencov_split` to extract the `*.gcda` files. Note that full build paths are used by the tools, so splitting needs to output relative to `/`.
- Use `geninfo` to post-process the raw data.
- Use `genhtml` to render the results as HTML.
- View the results in a browser.

```
xen.git/xen$ ssh root@host xencov read > coverage.dat
xen.git/xen$ ../tools/xencov_split coverage.dat --output-dir=/
xen.git/xen$ geninfo . -o cov.info
xen.git/xen$ genhtml cov.info -o cov/
xen.git/xen$ $BROWSER cov/index.html
```

Clang coverage

An example of how to view the data is as follows.

- Obtain the raw coverage data from the test host, and pull it back to the build working tree.
- Use `llvm-profdata` to post-process the raw data.
- Use `llvm-cov show` in combination with `xen-syms` from the build to render the results as HTML.
- View the results in a browser.

```
xen.git/xen$ ssh root@host xencov read > xen.profraw
xen.git/xen$ llvm-profdata merge xen.profraw -o xen.profdata
xen.git/xen$ llvm-cov show -format=html -output-dir=cov/ xen-syms -instr-profile=xen.
↪profdata
xen.git/xen$ $BROWSER cov/index.html
```

Full documentation on Clang's coverage capabilities can be found at: https://clang.llvm.org/docs/SourceBasedCodeCoverage.html

3.1.2 x86

How Xen Boots

This is an at-a-glance reference of Xen's booting capabilities and expectations.

Build

A build of xen produces `xen.gz` and optionally `xen.efi` as final artefacts.

- For BIOS, Xen supports the Multiboot 1 and 2 protocols.
- For EFI, Xen supports Multiboot 2 with EFI extensions, and native EFI64.
- For virtualisation, Xen supports starting directly with the PVH boot protocol.

Objects

To begin with, most object files are compiled and linked. This includes the Multiboot 1 and 2 headers and entrypoints, including the Multiboot 2 tags for EFI extensions. When `CONFIG_PVH_GUEST` is selected at build time, this includes the PVH entrypoint and associated ELF notes.

Depending on whether the compiler supports `__attribute__((__ms_abi__))` or not, either an EFI stub is included which nops/fails applicable setup and runtime calls, or full EFI support is included.

Protocols and entrypoints

All headers and tags are built in `xen/arch/x86/boot/head.S`

The Multiboot 1 headers request aligned modules and memory information. Entry is via the start of the binary image, which is the `start` symbol. This entrypoint must be started in 32bit mode.

The Multiboot 2 headers are more flexible, and in addition request that the image be loaded as high as possible below the 4G boundary, with 2M alignment. Entry is still via the `start` symbol as with MB1, and still in 32bit mode.

Headers for the EFI MB2 extensions are also present. These request that `ExitBootServices()` not be called, and register `__efi_mb2_start` as an alternative entrypoint, entered in 64bit mode.

If `CONFIG_PVH_GUEST` was selected at build time, an Elf note is included which indicates the ability to use the PVH boot protocol, and registers `__pvh_start` as the entrypoint, entered in 32bit mode.

xen.gz

The objects are linked together to form `xen-syms` which is an ELF64 executable with full debugging symbols. `xen.gz` is formed by stripping `xen-syms`, then repackaging the result as an ELF32 object with a single load section at 2MB, and `gzip`-ing the result. Despite the ELF32 having a fixed load address, its contents are relocatable.

Any bootloader which unzips the binary and follows the ELF headers will place it at the 2M boundary and jump to `start` which is the identified entry point. However, Xen depends on being entered with the MB1 or MB2 protocols, and will terminate otherwise.

The MB2+EFI entrypoint depends on being entered with the MB2 protocol, and will terminate if the entry protocol is wrong, or if EFI details aren't provided, or if EFI Boot Services are not available.

xen.efi

When a PEI-capable toolchain is found, the objects are linked together and a PE32+ binary is created. It can be run directly from the EFI shell, and has `efi_start` as its entry symbol.

Note: xen.efi does contain all MB1/MB2/PVH tags included in the rest of the build. However, entry via anything other than the EFI64 protocol is unsupported, and won't work.

Boot

Xen, once loaded into memory, identifies its position in order to relocate system structures. For 32bit entrypoints, this necessarily requires a call instruction, and therefore a stack, but none of the ABIs provide one.

Overall, given that on a BIOS-based system, the IVT and BDA occupy the first 5/16ths of the first page of RAM, with the rest free to use, Xen assumes the top of the page is safe to use.

CHAPTER 4

MISRA C coding guidelines

MISRA C rules and directive to be used as coding guidelines when writing Xen hypervisor code.

4.1 MISRA C rules for Xen

Note: **IMPORTANT** All MISRA C rules, text, and examples are copyrighted by the MISRA Consortium Limited and used with permission.

Please refer to https://www.misra.org.uk/ to obtain a copy of MISRA C, or for licensing options for other use of the rules.

The following is the list of MISRA C rules that apply to the Xen hypervisor.

It is possible that in specific circumstances it is best not to follow a rule because it is not possible or because the alternative leads to better code quality. Those cases are called "deviations". They are permissible as long as they are documented. For details, please refer to docs/misra/documenting-violations.rst

Other documentation mechanisms are work-in-progress.

The existing codebase is not 100% compliant with the rules. Some of the violations are meant to be documented as deviations, while some others should be fixed. Both compliance and documenting deviations on the existing codebase are work-in-progress.

The list below might need to be updated over time. Reach out to THE REST maintainers if you want to suggest a change.

Dir number	Severity	Summary	Notes
Dir 1.1	Required	Any implementation-defined behaviour on which the output of the program depends shall be documented and understood	
Dir 2.1	Required	All source files shall compile without any compilation errors	
Dir 4.7	Required	If a function returns error information then that error information shall be tested	
Dir 4.10	Required	Precautions shall be taken in order to prevent the contents of a header file being included more than once	
Dir 4.11	Required	The validity of values passed to library functions shall be checked	We do not have libraries in Xen (libfdt and others are not considered libraries from MISRA C point of view as they are imported in source form)
Dir 4.14	Required	The validity of values received from external sources shall be checked	

Rule number	Severity	Summary	Notes
Rule 1.1	Required	The program shall contain no violations of the standard C syntax and constraints, and shall not exceed the implementation's translation limits	We make use of several compiler extensions as documented by C-language-toolchain.rst
Rule 1.3	Required	There shall be no occurrence of undefined or critical unspecified behaviour	
Rule 1.4	Required	Emergent language features shall not be used	Emergent language features, such as C11 features, should not be confused with similar compiler extensions, which we use. When the time comes to adopt C11, this rule will be revisited.
Rule 2.1	Required	A project shall not contain unreachable code	
Rule 2.6	Advisory	A function should not contain unused label declarations	
Rule 3.1	Required	The character sequences /* and // shall not be used within a comment	
Rule 3.2	Required	Line-splicing shall not be used in // comments	
Rule 4.1	Required	Octal and hexadecimal escape sequences shall be terminated	

Continued on next page

Table 1 – continued from previous page

Rule number	Severity	Summary	Notes
Rule 4.2	Advisory	Trigraphs should not be used	
Rule 5.1	Required	External identifiers shall be distinct	The Xen characters limit for identifiers is 40. Public headers (xen/include/public/) are allowed to retain longer identifiers for backward compatibility.
Rule 5.2	Required	Identifiers declared in the same scope and name space shall be distinct	The Xen characters limit for identifiers is 40. Public headers (xen/include/public/) are allowed to retain longer identifiers for backward compatibility.
Rule 5.3	Required	An identifier declared in an inner scope shall not hide an identifier declared in an outer scope	Using macros as macro parameters at invocation time is allowed even if both macros use identically named local variables, e.g. max(var0, min(var1, var2))
Rule 5.4	Required	Macro identifiers shall be distinct	The Xen characters limit for macro identifiers is 40. Public headers (xen/include/public/) are allowed to retain longer identifiers for backward compatibility.
Rule 5.6	Required	A typedef name shall be a unique identifier	
Rule 6.1	Required	Bit-fields shall only be declared with an appropriate type	In addition to the C99 types, we also consider appropriate types enum and all explicitly signed / unsigned integer types.
Rule 6.2	Required	Single-bit named bit fields shall not be of a signed type	
Rule 7.1	Required	Octal constants shall not be used	

Continued on next page

Table 1 – continued from previous page

Rule number	Severity	Summary	Notes
Rule 7.2	Required	A "u" or "U" suffix shall be applied to all integer constants that are represented in an unsigned type	The rule asks that any integer literal that is implicitly unsigned is made explicitly unsigned by using one of the indicated suffixes. As an example, on a machine where the int type is 32-bit wide, 0x77777777 is signed whereas 0x80000000 is (implicitly) unsigned. In order to comply with the rule, the latter should be rewritten as either 0x80000000u or 0x80000000U. Consistency considerations may suggest using the same suffix even when not required by the rule. For instance, if one has: Original: f(0x77777777); f(0x80000000); one should do Solution 1: f(0x77777777U); f(0x80000000U); over Solution 2: f(0x77777777); f(0x80000000U); after having ascertained that "Solution 1" is compatible with the intended semantics.
Rule 7.3	Required	The lowercase character l shall not be used in a literal suffix	
Rule 7.4	Required	A string literal shall not be assigned to an object unless the object type is pointer to const-qualified char	All "character types" are permitted, as long as the string element type and the character type match. (There should be no casts.) Assigning a string literal to any object with type "pointer to const-qualified void" is allowed.
Rule 8.1	Required	Types shall be explicitly specified	

Continued on next page

Table 1 – continued from previous page

Rule number	Severity	Summary	Notes
Rule 8.2	Required	Function types shall be in prototype form with named parameters	
Rule 8.3	Required	All declarations of an object or function shall use the same names and type qualifiers	
Rule 8.4	Required	A compatible declaration shall be visible when an object or function with external linkage is defined	
Rule 8.5	Required	An external object or function shall be declared once in one and only one file	
Rule 8.6	Required	An identifier with external linkage shall have exactly one external definition	Declarations without definitions are allowed (specifically when the definition is compiled-out or optimized-out by the compiler)
Rule 8.8	Required	The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage	
Rule 8.10	Required	An inline function shall be declared with the static storage class	gnu_inline (without static) is allowed.
Rule 8.12	Required	Within an enumerator list the value of an implicitly-specified enumeration constant shall be unique	
Rule 8.14	Required	The restrict type qualifier shall not be used	
Rule 9.1	Mandatory	The value of an object with automatic storage duration shall not be read before it has been set	Rule clarification: do not use variables before they are initialized. An explicit initializer is not necessarily required. Try reducing the scope of the variable. If an explicit initializer is added, consider initializing the variable to a poison value.
Rule 9.2	Required	The initializer for an aggregate or union shall be enclosed in braces	
Rule 9.3	Required	Arrays shall not be partially initialized	{} is also allowed to specify explicit zero-initialization

Continued on next page

Table 1 – continued from previous page

Rule number	Severity	Summary	Notes
Rule 9.4	Required	An element of an object shall not be initialized more than once	
Rule 12.5	Mandatory	The sizeof operator shall not have an operand which is a function parameter declared as "array of type"	
Rule 13.6	Mandatory	The operand of the sizeof operator shall not contain any expression which has potential side effects	
Rule 13.1	Required	Initializer lists shall not contain persistent side effects	
Rule 14.1	Required	A loop counter shall not have essentially floating type	
Rule 16.7	Required	A switch-expression shall not have essentially Boolean type	
Rule 17.3	Mandatory	A function shall not be declared implicitly	
Rule 17.4	Mandatory	All exit paths from a function with non-void return type shall have an explicit return statement with an expression	
Rule 17.6	Mandatory	The declaration of an array parameter shall not contain the static keyword between the [ ]	
Rule 18.3	Required	The relational operators > >= < and <= shall not be applied to objects of pointer type except where they point into the same object	
Rule 19.1	Mandatory	An object shall not be assigned or copied to an overlapping object	Be aware that the static analysis tool Eclair might report several findings for Rule 19.1 of type "caution". These are instances where Eclair is unable to verify that the code is valid in regard to Rule 19.1. Caution reports are not violations.

Continued on next page

Table 1 – continued from previous page

Rule number	Severity	Summary	Notes
Rule 20.7	Required	Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses	
Rule 20.13	Required	A line whose first token is # shall be a valid preprocessing directive	
Rule 20.14	Required	All #else #elif and #endif preprocessor directives shall reside in the same file as the #if #ifdef or #ifndef directive to which they are related	
Rule 21.13	Mandatory	Any value passed to a function in <ctype.h> shall be representable as an unsigned char or be the value EOF	
Rule 21.17	Mandatory	Use of the string handling functions from <string.h> shall not result in accesses beyond the bounds of the objects referenced by their pointer parameters	
Rule 21.18	Mandatory	The size_t argument passed to any function in <string.h> shall have an appropriate value	
Rule 21.19	Mandatory	The pointers returned by the Standard Library functions localeconv, getenv, setlocale or, strerror shall only be used as if they have pointer to const-qualified type	
Rule 21.20	Mandatory	The pointer returned by the Standard Library functions asctime ctime gmtime localtime localeconv getenv setlocale or strerror shall not be used following a subsequent call to the same function	
Rule 21.21	Required	The Standard Library function system of <stdlib.h> shall not be used	
Rule 22.2	Mandatory	A block of memory shall only be freed if it was allocated by means of a Standard Library function	

Continued on next page

Table 1 – continued from previous page

Rule number	Severity	Summary	Notes
Rule 22.4	Mandatory	There shall be no attempt to write to a stream which has been opened as read-only	
Rule 22.5	Mandatory	A pointer to a FILE object shall not be dereferenced	
Rule 22.6	Mandatory	The value of a pointer to a FILE shall not be used after the associated stream has been closed	

CHAPTER 5

Miscellanea

5.1 Glossary

control domain A *domain*, commonly dom0, with the permission and responsibility to create and manage other domains on the system.

domain A domain is Xen's unit of resource ownership, and generally has at the minimum some RAM and virtual CPUs.

The terms *domain* and *guest* are commonly used interchangeably, but they mean subtly different things.

A guest is a single, end user, virtual machine.

In some cases, e.g. during live migration, one guest will be comprised of two domains for a period of time, while it is in transit.

domid The numeric identifier of a running *domain*. It is unique to a single instance of Xen, used as the identifier in various APIs, and is typically allocated sequentially from 0.

guest The term 'guest' has two different meanings, depending on context, and should not be confused with *domain*.

When discussing a Xen system as a whole, a 'guest' refer to a virtual machine which is the "useful output" of running the system in the first place (e.g. an end-user VM). Virtual machines providing system services, (e.g. the control and/or hardware domains), are not considered guests in this context.

In the code, "guest context" and "guest state" is considered in terms of the CPU architecture, and contrasted against hypervisor context/state. In this case, it refers to all code running lower privilege privilege level the hypervisor. As such, it covers all domains, including ones providing system services.

hardware domain A *domain*, commonly dom0, which shares responsibility with Xen about the system as a whole.

By default it gets all devices, including all disks and network cards, so is responsible for multiplexing guest I/O.

Index

C

control domain, **21**

D

domain, **21**
domid, **21**

G

guest, **21**

H

hardware domain, **21**