

Software User Manual

XM-ARM

Fent Innovative Software Solutions

July, 2017

Reference: 14-035-03.005.sum.04



This page is intentionally left blank.

DOCUMENT CONTROL PAGE

TITLE: Software User Manual: XM-ARM

AUTHOR/S: Fent Innovative Software Solutions

LAST PAGE NUMBER: 114

VERSION OF SOURCE CODE: XM-ARMv2.0.5 for the processor ARM CORTEX-A90

REFERENCE ID: 14-035-03.005.sum.04

SUMMARY: This guide describes the fundamental concepts and the features provided by the API of the XtratuM hypervisor.

DISCLAIMER: This documentation is currently under active development. Therefore, there are no explicit or implied warranties regarding any properties, including, but not limited to, correctness and fitness for purpose.

REFERENCING THIS DOCUMENT:

```
@techreport {14-035-03.005.sum.04,
  title = {Software User Manual: XM-ARM},
  author = { Fent Innovative Software Solutions},
  institution = {Fent Innovative Software Solutions, S.L.},
  number = {14-035-03.005.sum.04},
  year={July, 2017},
}
```

Copyright © July, 2017 Fent Innovative Software Solutions, S.L.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Changes:

Version	Date	Author	Comments
01	09/06/2015	Javier Coronel	Initial Software User Reference Manual for XM-ARMv2.0.0
02	14/01/2016	Manuel Muñoz and Javier Coronel	Update xml example to multicore Add "make defconfig" description Add XM_resume_imm_partition hypercall Add XM_switch_imm_sched_plan hypercall Update Warm Reset description Update Configuration Section Update memory protection mechanisms Update requirements Update processor management Update section 5.2 XAL development environment Add Hypervisor Feature configuration

Version	Date	Author	Comments
			Add “XM_HYP_FEAT_FPGA_PART_ACCESS” feature. Changes for XM-ARMv2.0.3
03	09/09/2016	Manuel Muñoz and Javier Coronel	Add memory XM memory map description. Add memory and peripheral access description. Add UART Clock configuration. Changes for XM-ARMv2.0.4
04	01/06/2017	Yolanda Valiente and Javier Coronel	Add L1 and L2 cache supported policies. Section 2.6. Add snoop control unit (SCU) description. Section 2.14.1. Add branch prediction description. Section 2.14.2. Add memory map access table. Section 5.15.2. Add parity errors description. Section 5.9.2. Add timer configuration allowed frequency values restriction. Section 8.2.4. Add UART clock allowed frequency values. Section 8.1. Changes for XM-ARMv2.0.5

Contents

Preface	xi
1 Introduction	1
1.1 History	2
2 XtratuM Architecture	5
2.1 System operation	6
2.2 Partition operation	7
2.3 System partitions	8
2.4 Names and identifiers	9
2.5 Partition scheduling	9
2.5.1 Multiple scheduling plans	12
2.6 Memory management	13
2.6.1 XtratuM memory map	14
2.7 Cache management	14
2.8 Inter-partition communications (IPC)	14
2.8.1 Multicore implications	16
2.9 Health monitor (HM)	16
2.9.1 HM Events	18
2.9.2 HM Actions	19
2.9.3 HM Configuration	19
2.9.4 HM notification	19
2.9.5 Multicore implications	20
2.10 Access to devices	21
2.11 Exceptions	21
2.11.1 Interpartition virtual interrupt management	22
2.11.2 Multicore implications	23
2.12 Traces	23
2.12.1 Multicore implications	23

2.13	Clocks and timers	23
2.14	Symmetric Multi-Processing (SMP) architecture	24
2.14.1	Snoop Control Unit (SCU)	25
2.14.2	Branch Prediction	25
2.15	Inter-Partition Virtual Interrupt (IPVI)	25
2.16	Status	26
2.17	Summary	26
3	Development Process Overview	29
3.1	Development at a glance	30
3.2	Building XtratuM	31
3.3	System configuration	32
3.4	Compiling partition code	33
3.5	Passing parameters to the partitions: customisation files	35
3.6	Building the final system image	35
4	Building XtratuM	37
4.1	Development environment	37
4.2	Compile XtratuM Hypervisor	37
4.3	Generating a binary distribution	39
4.4	Installing a binary distribution	41
4.5	Compile the Hello World! partition	41
5	Partition Programming	43
5.1	Partition definition	43
5.2	XAL development environment	44
5.2.1	XAL Partition Initialization	45
5.2.2	Facilities provided by XAL	45
5.2.3	XAL Partition Example	45
5.3	Partition reset	47
5.4	System reset	47
5.5	Scheduling	47
5.5.1	Slot identification	47
5.5.2	Managing scheduling plans	48
5.6	Console output	48
5.7	Inter-partition communication	49
5.7.1	Message notification	49
5.8	Peripheral programming	50

5.8.1	Direct Memory-mapped I/O	50
5.8.2	Device is accessed by IOPorts	50
5.8.3	FPGA Partition Access	51
5.8.4	PMU Partition Access	53
5.9	Exceptions	53
5.9.1	Interrupts/Fast interrupts	53
5.9.2	Parity errors	54
5.10	Clock and timer services	55
5.10.1	Execution time clock	55
5.11	Processor management	55
5.11.1	Current Program Status Register	55
5.11.2	Processor Modes	55
5.12	Synchronization	56
5.13	Tracing	56
5.13.1	Trace messages	56
5.13.2	Reading traces	56
5.14	System and partition status	57
5.15	Memory management	58
5.15.1	Partition Memory Map	58
5.15.2	Memory access restrictions	58
5.15.3	Configuration	60
5.16	Releasing the processor	60
5.17	Partition customisation files	60
5.18	The object interface	61
6	Binary Interfaces	63
6.1	Data representation	63
6.2	Hypercall mechanism	64
6.3	Assembly programming	64
6.4	Executable formats overview	65
6.5	Partition ELF format	66
6.5.1	Partition image header	66
6.5.2	Partition control table (PCT)	67
6.6	XEF format	69
6.6.1	Compression algorithm	71
6.7	Container format	71
7	Booting	75

7.1 Partition booting	76
8 Configuration	77
8.1 XtratuM source code configuration (menuconfig)	77
8.2 Hypervisor configuration file (XM_CF)	79
8.2.1 Data representation and XPath syntax	79
8.2.2 The root element: /SystemDescription	81
8.2.3 The /SystemDescription/XMHypervisor element	81
8.2.4 The /SystemDescription/HwDescription element	82
8.2.5 The /SystemDescription/ResidentSw element	83
8.2.6 The /SystemDescription/PartitionTable/Partition element	83
8.2.7 The /SystemDescription/Channels element	86
9 Tools	87
9.1 XML configuration parser (xmcparser)	87
9.1.1 xmcparser	87
9.2 ELF to XEF (xmeformat)	88
9.2.1 xmeformat	88
9.3 Container builder (xmpack)	90
9.3.1 xmpack	90
9.4 Bootable image creator (rswbuild)	91
9.4.1 rswbuild	91
10 Security issues	93
10.1 Invoking a hypercall from libXM	93
10.2 Preventing covert/side channels due to scheduling slot overrun	93
A XML Schema Definition	95
A.1 XML Schema file	95
A.2 Configuration file example	103
GNU Free Documentation License	105
1. APPLICABILITY AND DEFINITIONS	105
2. VERBATIM COPYING	106
3. COPYING IN QUANTITY	106
4. MODIFICATIONS	107
5. COMBINING DOCUMENTS	108
6. COLLECTIONS OF DOCUMENTS	108
7. AGGREGATION WITH INDEPENDENT WORKS	108

8. TRANSLATION	109
9. TERMINATION	109
10. FUTURE REVISIONS OF THIS LICENSE	109
ADDENDUM: How to use this License for your documents	109
Applicable and reference documents	111
Glossary of Terms and Acronyms	113

This page is intentionally left blank.

Preface

The target readers for this document are software developers who need to use the services of XtratuM directly. The reader is expected to have an in-depth knowledge of the ARM CORTEX-A9 architecture and experience in programming device drivers. It is also advisable to have some knowledge of the ARINC-653 and other related standards.

Typographical conventions

The following typographical conventions are used in this document:

- **typewriter:** used in assembler and C code examples, and to show the output of commands.
- *italic:* used to introduce new terms.
- **bold face:** used to emphasize or highlight a word or paragraph.

Code

Code examples are printed inside a box as shown in the following example:

```
xm_s32_t XM_hm_open(void) {  
    if (!(libXmParams.partCtrlTab->flags&XM_PART_SYSTEM))  
        return XM_PERM_ERROR;  
  
    return XM_OK;  
}
```

Listing 1: Sample code

Caution sign

The caution sign stresses information that is critical to the integrity or continuity of the system.



Support

Fent Innovative Software Solutions (FentISS)
Camino de vera s/n
CP: 46022

Valencia, Spain

The official XtratuM web site is: <http://www.fentiss.com>

Chapter 1

Introduction

This document describes the XtratuM hypervisor, and how to write applications to be executed as XtratuM partitions.

A hypervisor is a layer of software that provides one or more virtual execution environments for partitions. Although virtualisation concepts have been employed since the 60's (IBM 360), the application of these concepts to the server, desktop, and recently to the embedded and real-time computer segments, is relatively new. There have been some attempts, in the desktop and server markets, to standardise “how” a hypervisor should operate, but research and the market are not mature enough. In fact, there is still not a common agreement on the terms used to refer to some of the new objects introduced. Check the glossary [A.2](#) for the exact meaning of the terms used in this document.

In the case of embedded systems and, in particular, in avionics, the ARINC-653 standard defines a partitioning system. Although the ARINC-653 standard was not designed to describe how a hypervisor has to operate, some parts of the APEX model of ARINC-653 are quite close to the functionality provided by a hypervisor.

Though during the early development of XtratuM, we adapted the XtratuM API and internals operations to resemble the ARINC-653 standard, it is not our intention to convert XtratuM into an ARINC-653 compliant system. ARINC-653 relies on the idea of a “*separation kernel*”, which basically consists in extending and enforcing the isolation between processes or a group of processes. ARINC-653 defines both the API and operation of the partitions, but also how the threads or processes are managed inside each partition. It provides a complete APEX.

In a bare-metal hypervisor, and in particular in XtratuM, a partition is a *virtual computer* rather than a group of strongly isolated processes. When multi-threading (or tasking) support is needed in a partition, then an operating system or a run-time support library has to provide support to the application threads. In fact, it is possible to run a different operating system on each XtratuM partition.

It is important to point out that XtratuM is a bare-metal hypervisor with extended capabilities for highly critical systems. XtratuM provides a raw (close to the native hardware) virtual execution environment, rather than a full featured one. Therefore, **although XtratuM by itself can not be compatible with the ARINC-653 standard, the philosophy of the ARINC-653 has been employed when applicable.**

This document is organised as follows:

Chapter [2](#) describes the XtratuM architecture describing how the partitions are organised and scheduled; also, an overview of the XtratuM services is presented.

Chapter [3](#) outlines the development process on XtratuM: roles, elements, etc.

Chapter 4 describes the compilation process, which involves several steps to finally obtain a binary code which has to be loaded in the embedded system.

35 The goal of chapter 5 is to provide a view of the API provided by XtratuM to develop applications to be executed as partitions. The chapter puts more emphasis in the development of bare-applications than applications running on a real-time operating system.

Chapter 6 deals with the concrete structure and internal format of the different components involved in the system development: system image, partition format, partition tables. The chapter ends with the
40 description of the hypercall mechanism.

Chapter 7 and 8 detail the booting process and the configuration elements of the system, respectively. Finally, chapter 8 provides information about the preliminary tools developed to analyse system configuration schema files (XML format) and generate the appropriate internal structures to configure XtratuM for a specific payload.

1.1 History

45 The term XtratuM derives from the word “stratum”. In geology and related fields it means:

Layer of rock or soil with internally consistent characteristics that distinguishes it from contiguous layers.

In order to stress the tight relation with Linux and the open source the “S” was replaced by “X”. XtratuM would be the first layer of software (the one closer to the hardware), which provides a rock-solid basis
50 for the rest of the system.

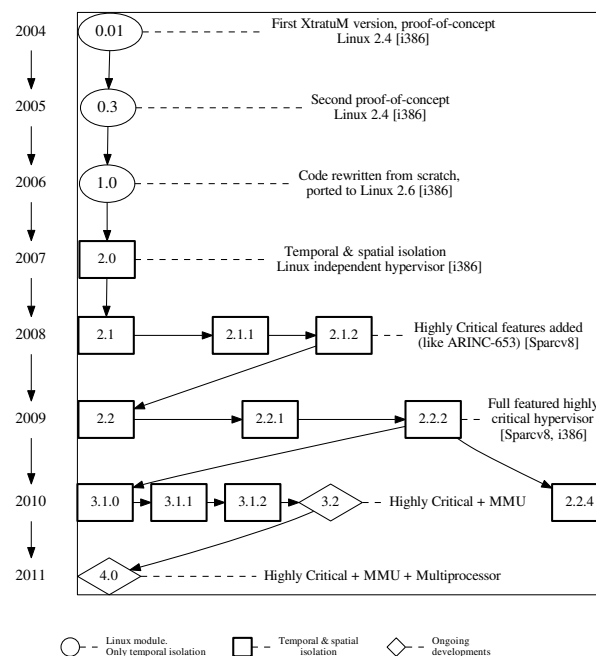


Figure 1.1: XtratuM evolution.

The first version of XtratuM (1.0) was initially developed to meet the requirements of a hard real-time system. The main goal of XtratuM 1.0 was to guarantee the temporal constraints for the real-time partitions. Other characteristics of this version are:

- The first partition shall be a modified version of Linux.

- Partition code has to be loaded dynamically. 55
- There is not a strong memory isolation between partitions.
- Linux is executed in processor supervisor mode.
- Linux is responsible for booting the computer.
- Fixed priority partition scheduling.

XtratuM 2.0 was a completely new redesign and implementation. This new version had nothing in common with the first one but the name. It was truly a hypervisor with both, spatial and temporal isolation. This version was developed for the x86 architecture but it was never released. 60

XtratuM 2.1 was the first porting to the LEON2 processor, and several safety critical features were added. Just to mention the most relevant features:

- Bare-metal hypervisor. 65
- Employs para-virtualisation techniques.
- A hypervisor designed for embedded systems: some devices can be directly managed by a designated partition.
- Strong temporal isolation: fixed cyclic scheduler.
- Strong spatial isolation: all partitions are executed in the user mode of the processor, and do not share memory. 70
- Resource allocation via a configuration table.
- Robust communication mechanisms (ARINC sampling and queuing ports).

Version 2.1 was a prototype to evaluate the capabilities of the LEON2 processor to support a hypervisor system. 75

XtratuM 2.2 was a more mature hypervisor on the LEON2 processor. This version has most of the final functionality.

XtratuM 3.1 introduces several changes. The main changes are:

- Audit events have been added as the XtratuM tracing subsystem.
- Virtual interrupt subsystem has been rewritten from scratch. 80
- Cache instruction burst fetch capability can be either enabled or disabled during the XtratuM building process.
- Cache snoop feature can be either enabled or disabled during the XtratuM building process.
- Several partitions can be built with the same virtual addresses.
- Hypercalls behaviour is more robust. 85

XtratuM 3.2 introduces several changes. The main changes are:

- Any exception caused by a partition when the processor is in supervisor mode causes a partition unrecoverable error.
- A cache flush is executed always in order to guarantee that a partition is unable to retrieve XM information. This operation is forced 90

XtratuM 3.5 to 3.9 introduces several changes. The main changes are:

- Multiprocessor support. Introduction of the definition of virtual CPUs.
- Introduce the ARM architecture.

The current development version is XM-ARMv2.0. This version is still under active development.

⁹⁵ In what follows, the name XtratuM will be used to refer to the version XM-ARMv2.0.x of the hypervisor.

Chapter 2

XtratuM Architecture

This chapter introduces the architecture of XtratuM.

The concept of partitioned software architectures was developed to address security and safety issues. The central design criteria involves isolating modules of the system into *partitions*. Temporal and spatial isolation are the key aspects in a partitioned system. Based on this approach, the Integrated Modular Avionics (IMA) is a solution allowed by the Aeronautic Industry to manage the increment of the functionalities of the software maintaining the level of efficiency.

XtratuM is a bare-metal hypervisor designed to achieve temporal and spatial partitioning for safety critical applications. Figure 2.1 shows the complete architecture.

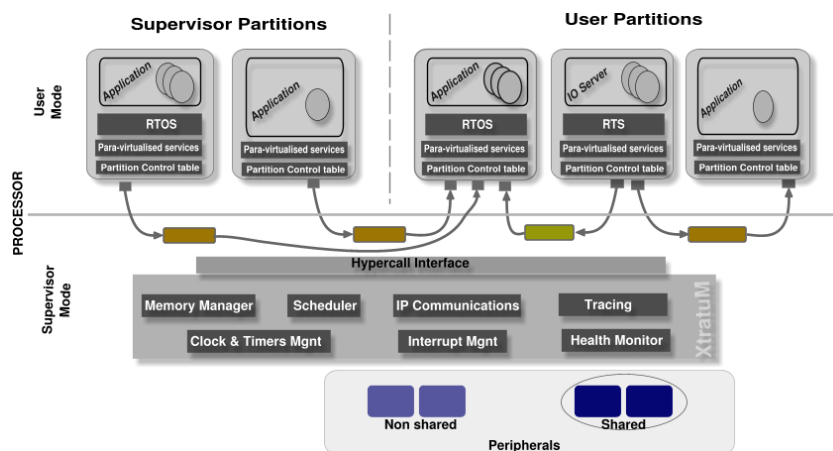


Figure 2.1: XtratuM architecture.

The main components of this architecture are:

- Hypervisor: XtratuM provides virtualisation services to partitions. It is executed in supervisor mode of the processor and virtualises the CPU, memory, interrupts, and some specific peripherals. The internal XtratuM architecture includes the following components:
 - Memory management: XtratuM provides a memory model for the partitions enforcing the spatial isolation. It uses the hardware mechanisms to guarantee the isolation.
 - Scheduling: Partitions are scheduled by using a cyclic scheduling policy.
 - Interrupt management: Interrupts could be configured to be used by the partitions. But exclusively by one of them. XtratuM provides an interrupt model to the partitions that extends the concept of processor interrupts by adding 32 additional sources of interrupt (events).

– Clock and timer management.

– IP communication: Inter-partition communication is related to the communications between two partitions or between a partition and the hypervisor. XtratuM implements a message passing model which highly resembles the one defined in the ARINC-653. A communication channel is the logical path between one source and one or more destinations. Two basic transfer modes are provided: sampling and queuing. Partitions can access to channels through access points named ports. The hypervisor is responsible for encapsulating and transporting messages.

– Health monitor: The health monitor is the part of XtratuM that detects and reacts to anomalous events or states. Some of these event are captured by the partitions which must communicate to XtratuM. The purpose of the HM is to discover the errors at an early stage and try to solve or confine the faulting subsystem in order to avoid or reduce the possible consequences.

– Tracing facilities: XtratuM provides a mechanism to store and retrieve the traces generated by partitions and XtratuM itself. Traces can be used for debugging, during the development phase of the application, but also to log relevant events or states during the production phase.

- API: Defines the para-virtualised services provided by XtratuM. The access to these services is provided through of *hypercalls*.

- Partitions: A partition is an execution environment managed by the hypervisor which uses the virtualised services. Each partition consists of one or more concurrent processes (concurrency must be implemented by the operating system of each partition because it is not directly supported by XtratuM), that share access to processor resources based upon the requirements of the application. The partition code can be: an application compiled to be executed on a bare-machine; a real-time operating system (or runtime support) and its applications; or a general purpose operating system and its applications.

Partitions need to be *virtualised* to be executed on top of a hypervisor. Depending on the type of execution environment, the virtualisation implications in each case can be summarised as:

Bare application : The application has to be virtualised using the services provided by XtratuM. The application is designed to run directly on the hardware and therefore it has to be adapted to use the para-virtualised services.

Mono-core Operating system application : When the application runs on top of a (real-time) operating system, it uses the services provided by the operating system and does not need to be directly virtualised. However, the operating system has to deal with the virtualisation and be virtualised (ported on top of XtratuM).

Multi-core Operating system application : A guest OS can use multiple virtual CPU to support multicore applications

2.1 System operation

The system's states and its transitions are shown in figure 2.2.

At boot time, the resident software loads the image of XtratuM in main memory and transfers control to the entry point of XtratuM. The period of time between starting from the entry point and the execution of the first partition is defined as **boot** state. In this state, the scheduler is not enabled and the partitions are not executed (see chapter 7).

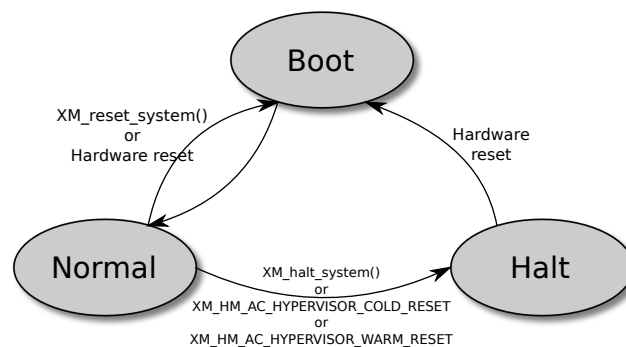


Figure 2.2: System's states and transitions.

At the end of the boot sequence, the hypervisor is ready to start executing partition code. The system changes to **normal** state and the scheduling plan is started. Changing from boot to normal state is performed automatically (the last action of the set up procedure).

The system can switch to **halt** state by the health monitoring system in response to a detected error or by a *system partition* invoking the service `XM_halt_system()`. In the halt state: the scheduler is disabled, the hardware interrupts are disabled, and the processor enters in an endless loop. The only way to exit from this state is via an external hardware reset.

It is possible to perform a warm or cold (hardware reset) system reset by using the hypercall (see `XM_reset_system()`). On a warm reset, the system increments the reset counter, and a reset value is passed to the new rebooted system. On a cold reset, no information about the state of the system is passed to the new rebooted system.

2.2 Partition operation

Once XtratuM is in normal state, partitions are started. The partition's states and transitions are shown in figure 2.3.

On start-up each partition is in boot state. It has to prepare the virtual machine to be able to run the applications¹: it sets up a standard execution environment (that is, initialises a correct stack and sets up the virtual processor control registers), creates the communication ports, requests the hardware devices (I/O ports and interrupt lines), etc., that it will use. Once the partition has been initialised, it changes to normal mode.

The partition receives information from XtratuM about the previous executions, if any.

From the hypervisor point of view, there is no difference between the boot state and the normal state. In both states the partition is scheduled according to the fixed plan, and has the same capabilities. Although not mandatory, it is recommended that the partition emits a partition's state-change event when changing from boot to normal state.

The normal state is subdivided in three sub-states:

Ready The partition is ready to execute code, but it is not scheduled because it is not in its time slot.

Running The partition is being executed by the processor.

Idle If the partition does not want to use the processor during its allocated time slot, it can relinquish the processor and wait for an interrupt or for the next time slot (see `XM_idle_self()`).

¹We will consider that the partition code is composed of an operating system and a set of applications.

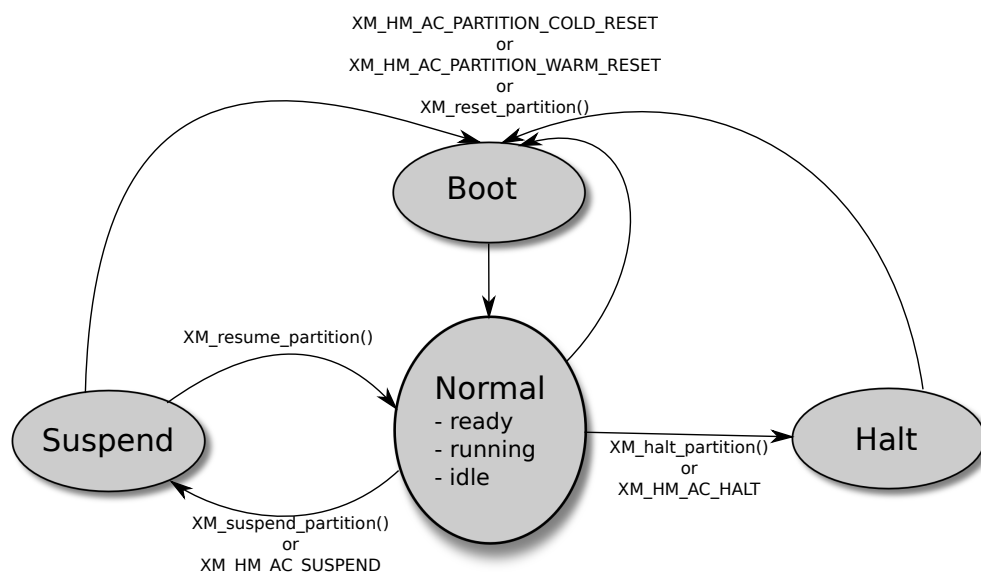


Figure 2.3: Partition states and transitions.

A partition can halt itself or be halted by a system partition. In the halt state, the partition is not selected by the scheduler and the time slot allocated to it is left idle (it is not allocated to other partitions). All resources allocated to the partition are released. It is not possible to return to normal state.

In suspended state, a partition will not be scheduled and interrupts are not delivered. Interrupts raised while in suspended state are left pending. If the partition returns to normal state, then pending interrupts are delivered to the partition. The partition can return to ready state if requested by a system partition by calling `XM_resume_partition()` or `XM_resume_imm_partition()` hypercall.

2.3 System partitions

XtratuM defines two types of partitions: *normal* and *system*. System partitions are allowed to manage and monitor the state of the system and other partitions. Some hypercalls cannot be called by a normal partition or have restricted functionality.

Note that system partition rights are related to the capability to manage the system, and not to the capability to access directly to the native hardware or to break the isolation: a system partition is scheduled as a normal partition; and it can only use the resources allocated to it in the configuration file.

Table 2.1 shows the list of hypercalls reserved for system partitions. A hypercall labeled as “partial” indicates that a normal partition can invoke it if a system reserved service is not requested.

A partition has system capabilities if the `/System_Description/Partition_Table/Partition/@flags` attribute contains the flag “system” in the XML configuration file. Several partitions can be defined as system partition.

Hypercall	System
<code>XM_get_gid_by_name</code>	Partial
<code>XM_get_partition_status</code>	Partial
<code>XM_get_system_status</code>	Yes
<code>XM_halt_partition</code>	Partial
<code>XM_halt_system</code>	Yes
<code>XM_hm_read</code>	Yes
<code>XM_hm_status</code>	Yes
<code>XM_memory_copy</code>	Partial
<code>XM_reset_partition</code>	Partial
<code>XM_reset_system</code>	Yes
<code>XM_resume_imm_partition</code>	Yes
<code>XM_resume_partition</code>	Yes
<code>XM_shutdown_partition</code>	Partial
<code>XM_suspend_partition</code>	Partial
<code>XM_switch_imm_sched_plan</code>	Yes
<code>XM_switch_sched_plan</code>	Yes
<code>XM_trace_read</code>	Yes
<code>XM_trace_status</code>	Yes

Table 2.1: List of system reserved hypercalls.

2.4 Names and identifiers

Each partition is globally identified by a unique identifier *id*. Partition identifiers are assigned by the integrator in the `XM.CF` file. XtratuM uses this identifier to refer to partitions. System partitions use partition identifiers to refer to the target partition. The “C” macro `XM_PARTITION_SELF` can be used by a partition to refer to itself.

These *ids* are used internally as indexes to the corresponding data structures². The first identifier (*id*) of each object group shall start in zero and the next *id*’s shall be consecutive. It is mandatory to follow this order in the `XM.CF` file.

The attribute *name* of a partition is a human readable string. This string shall contain only the following set of characters: upper and lower case letters, numbers and the underscore symbol. It is advisable not to use the same name on different partitions. A system partition can get the name of another partition by consulting the status object of the target partition.

In order to avoid name collisions, all the hypercalls of XtratuM contain the prefix “XM”. Therefore, the prefix “XM”, both in upper and lower case, is reserved.

2.5 Partition scheduling

XtratuM schedules partitions in a fixed, cyclic basis (ARINC-653 scheduling policy). This policy ensures that one partition cannot use the processor for longer than scheduled to the detriment of the other partitions. The set of *time slots* allocated to each partition is defined in the `XM.CF` configuration file during the design phase. Each partition is scheduled for a time slot defined as a start time and a duration. Within a time slot, XtratuM allocates the processor to the partition.

If there are several concurrent activities in the partition, the partition shall implement its own scheduling algorithm. This two-level scheduling scheme is known as *hierarchical scheduling*. XtratuM is not

²For efficiency and simplicity reasons.

aware of the scheduling policy used internally on each partition.

225 In general, a cyclic plan consists of a major time frame (MAF) which is periodically repeated. The MAF is typically defined as the least common multiple of the periods of the partitions (or the periods of the threads of each partition, if any).

	Name	Period	WCET	Util %
Partition 1	System Mngmt	100	20	20
Partition 2	Flight Control	100	10	10
Partition 3	Flight Mngmt	100	30	30
Partition 4	IO Processing	100	20	20
Partition 5	IHVM	200	20	10

(a) Partition set.

	Start	Dur.	Start	Dur.	Start	Dur.	Start	Dur.
Partition 0	0	20	100	20				
Partition 1	20	10	120	10				
Partition 2	40	30	140	30				
Partition 3	30	10	70	10	130	10	170	10
Partition 4	180	20						

(b) Detailed execution plan.

Table 2.2: Partition definition.

For instance, consider the partition set of figure 2.2a, its hyper-period is 200 time units (milliseconds) and has a CPU utilisation of the 90%. The execution chronogram is depicted in figure 2.4. One of the possible cyclic scheduling plans can be described, in terms of start time and duration, as it is shown in the table 2.2b.

This plan has to be specified in the configuration file. An XML file describing this schedule is shown below.

```

235 <?xml version="1.0"?>
    <Processor frequency="80Mhz" id="0">
        <Sched>
            <CyclicPlan>
                <Plan majorFrame="1s">
240             <Slot duration="20ms" id="0" partitionId="0" start="0ms"/>
                <Slot duration="10ms" id="1" partitionId="1" start="20ms"/>
                <Slot duration="10ms" id="2" partitionId="0" start="30ms"/>
                <Slot duration="30ms" id="3" partitionId="2" start="40ms"/>
                <Slot duration="10ms" id="4" partitionId="1" start="70ms"/>
245             <Slot duration="20ms" id="5" partitionId="0" start="100ms"/>
                <Slot duration="10ms" id="6" partitionId="1" start="120ms"/>
                <Slot duration="10ms" id="7" partitionId="0" start="130ms"/>
                <Slot duration="30ms" id="8" partitionId="2" start="140ms"/>
                <Slot duration="10ms" id="9" partitionId="1" start="170ms"/>
250             <Slot duration="20ms" id="10" partitionId="0" start="180ms"/>
                </Plan>
            </CyclicPlan>
        </Sched>
    </Processor>
255

```

Listing 2.1: Plan example

One important aspect in the design of the XtratuM hypervisor scheduler is the consideration of the overhead caused by the partition's context switch. Figure 2.5 shows the implications of this issue.

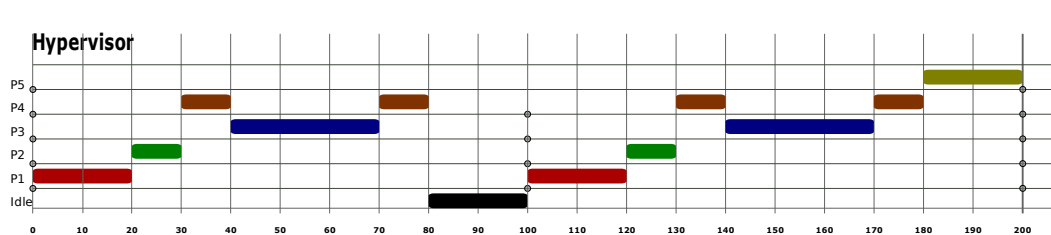


Figure 2.4: Scheduling example.

Subfigure 2.5a shows the context switch between partitions 1 and 2. To execute the partition, XtratuM saves the partition 1's context and loads the partition 2's context.

XtratuM scheduling design tries to adjust as much as possible the beginning of the execution to the specified start time of the slot. To do that, when a slot is scheduled, XtratuM programs a timer with the duration of the slot minus the temporal cost of the complete context switch (load and save the context). Subfigure 2.5b shows this situation.

260

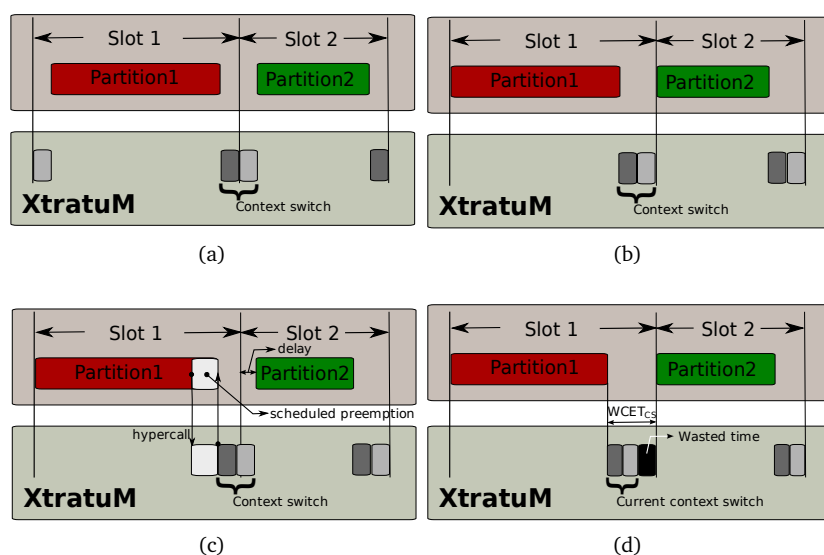


Figure 2.5: XtratuM context switch analysis.

However, the scenario depicted in subfigure 2.5c can occur. In this case, just before the duration timer expiration the partition invokes a hypercall. When the hypercall finishes, the timer interrupt is detected and the context switch is done at that time. This situation can introduce some small delay in the beginning of partition of the next scheduling time slot.

265

Figure 2.5d details what should be the value of the considered cost of the context switch. If the duration of the context switch is assumed as the worst case execution time of the context switch ($WCET_{CS}$), a situation like the one shown in figure 2.5d may happen. In this example, the cost of the context switch is less than its $WCET_{CS}$ and, as a consequence, an idle time has to be introduced to start the execution of the partition at the specified time.

270

XtratuM copes this situation by implementing the following algorithm:

- When a partition is scheduled, a timer (Scheduler Timer, ST) is armed with a value that considers the absolute start time of the next time slot, and the best case execution time of the context switch ($BCET_{CS}$).

275

- Two situations can introduce a small delay to the effective starting of the slot:
 1. The actual cost of the context switch is larger than the $BCET_{CS}$. In this case, the execution will start with a delay that is $WCET_{CS} - BCET_{CS}$.
 2. The ST expires while a hypercall is under execution. XtratuM will carry out the context switch when the current hypercall is finished, which delays the context switch. The worst case situation corresponds to the hypercall with longer execution time: $WCET_{HC}$.

Both previous situations can occur simultaneously. So, the worst case delay can be estimated as $(WCET_{CS} - BCET_{CC}) + WCET_{HP}$.

The cost of a context switch (both: $WCET_{CS}$ and $BCET_{CS}$) and all hypercalls have been evaluated and the worst case situation has been identified. In the document “*Volume 3: Testing and Evaluation*” it is provided a deep analysis of the hypercalls. The integrator must consider the worst case execution time of the used hypercalls and the partition context switch to forecast the slot duration considering the hypercalls used in the partition and the XtratuM configuration parameters.

2.5.1 Multiple scheduling plans

In some cases, a single scheduling plan may be too restrictive. For example:

- Depending on the guest operating system, the initialisation can require a certain amount of time and can vary significantly. If there is a single plan, the initialisation of each partition can require a different number of slots due to the fact that the slot duration has been designed considering the operational mode. This implies that a partition can be executing operational work whereas others are still initialising its data.
- The system can require to execute some maintenance operations. These operation can require allocating other resources different from the ones required during the operational mode.

In order to deal with these issues, XtratuM provides multiple scheduling plans that allows to reallocate the timing resources (the processor) in a controlled way. In the scheduling theory this process is known as mode changes. Figure 2.6 shows how the modes have been considered in the XtratuM scheduling.

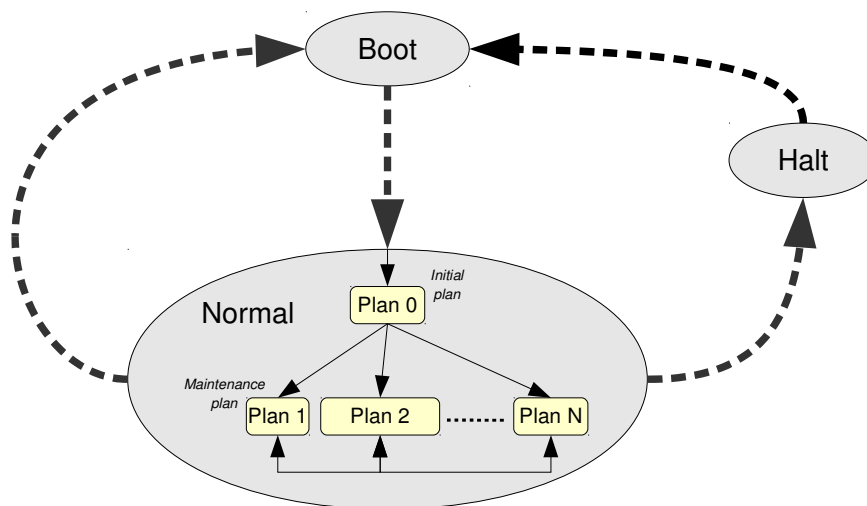


Figure 2.6: Scheduling modes.

The scheduler (and so, the plans) is only active while the system is in *normal* mode. Plans are defined in the `XM_CF` file and identified by a identifier. Some plans are reserved or have a special meaning:

Plan 0: *Initial plan.* The system selects this plan after a system reset. The system will be in plan 0 until a plan change is requested.

It is not legal to switch back to this plan. That is, this plan is only executed as a consequence of a system reset (software or hardware).

Plan 1: *Maintenance plan.* This plan can be activated in two ways:

- As a result of the health monitoring action `XM_HM_AC_SWITCH_TO_MAINTENANCE`. The plan switch is done immediately.
- Requested by a system partition. The plan switch occurs at the end the current plan or immediately depending on the called service.

It is advisable to allocate the first slot of this plan to a system partition, in order to start the maintenance activity as soon as possible after the plan switch. Once the maintenance activities have been completed, it is responsibility of a system partition to switch to another plan (if needed).

A system partition can also request a switch to this plan.

Plan x (x>1): Any plan greater than 1 is user defined. A system partition can switch to any of these defined plan at any time.

Switching scheduling plans

When a plan switch is requested by a system partition through one of these hypercalls:

XM.switch_sched_plan the plan switch is not immediate; all the slots of the current plan will be completed, and the new plan will be started at the end of the current one.

XM.switch_imm_sched_plan The current slot is terminated, and the new Plan is started immediately. This type of change will interrupt the execution of all the partition on each core in the system and it will force the beginning of a new MAF for the new scheduling plan.

The plan switch that occurs as a consequence of the `XM_HM_AC_SWITCH_TO_MAINTENANCE` action is synchronous. The current slot is terminated, and the Plan 1 is started immediately.

2.6 Memory management

Partitions and XtratuM core can be allocated at defined memory areas specified in the `XM.CF`.

A partition define several memory areas. Each memory area can define some attributes or rights to permit to other partitions to access to their defined areas or allow the cache management. The following attributes area allowed:

unmapped Allocated to the partition, but not mapped by XtratuM in the page table.

mappedAt It allows to allocate this area from a virtual address.

read-only The area is write-protected to the partition.

uncacheable Memory cache is disabled.

When the memory cache is enabled for the partition memory area (uncacheable flag is not set), the used L1 cache policy is the one defined in the configuration parameter [Hypervisor - Select L1 cache policy for partitions] during the XtratuM building process. Additionally, the L2 cache can be either

enabled or disabled, and the L2 policy can be configured through the configuration menu [Processor - Select L2 cache policy]. For detailed information about these configuration parameters, please refer to Section 8.1.

340 When the MMU is used, a partition can be allocated to a specified physical memory but using a virtual memory address (`mappedAt` attribute). In this case, the partition will be loaded in the physical address but this memory will be mapped to the virtual address specified.

2.6.1 XtratuM memory map

345 The hypervisor is placed in memory into 2 different memory areas one for executable code and the other for data. Where these memory areas are placed is defined during the hypervisor configuration and building process. Details about the configuration of these memory areas are explained in Section 8.1.

For each of these memory areas XtratuM reserves some bounding regions by rounding their real size. These areas cannot be used by partitions. The sizes of these memory areas are shown in Table 2.3.

	Physical	Virtual
XM Executable Code	1MB	16MB
XM Data	Real Size	16MB

Table 2.3: XM Memory Map Sizes.

350 The L1 cache policy for XtratuM executable Code and data sections are configured in the configuration parameters [Hypervisor - Select XM Code section cache policy] and [Hypervisor - Select XM Data section cache policy] respectively (see Section 8.1).

2.7 Cache management

In order to allow a more fine grain analysis of partition code, XtratuM allows to define the memory areas allocated to any partition as cacheable or non cacheable.

It is responsibility of the integrator to define the appropriated cache feature taking into account the system characteristics.

355 Additionally, partition code can also act on the cache when it is allowed in the XM.CF. XtratuM defines a hypercall `XM_set_cache_state()` to perform an action on the cache.

The hypercall requires two parameters:

- type of the cache: data or instruction cache
- operation: the actions to be done are: flush, freeze and unfreeze.

360 Cache state is not preserved longer than the current slot. It means that each time a new slot is scheduled the state of the cache is cacheable or non cacheable according the XM.CF definition.

2.8 Inter-partition communications (IPC)

Inter-partition communications are communications between two partitions. XtratuM implements a message passing model which highly resembles the one defined in the ARINC-653 standard. A message is a variable³ block of data. A message is sent from a source partition to one or more destination

³XtratuM defines the maximum length of a message.

partitions. The data of a message is transparent to the message passing system.

A communication channel is the logical path between one source and one or more destinations. Partitions can access to channels through access points named ports. The hypervisor is responsible for encapsulating and transporting messages that have to arrive to the destination(s) unchanged. At partition level, messages are atomic entities: either the whole message is received or nothing is received. Partition developers are responsible for agreeing on the format (data types, endianness, padding, etc.).

Channels, ports, maximum message sizes and maximum number of messages (queuing ports) are entirely defined in the configuration files (see section 8).

XtratuM provides two basic transfer modes: *sampling* and *queuing*.

Sampling port: It provides support for broadcast, multicast and unicast messages. No queuing is supported in this mode. A message remains in the source port until it is transmitted through the channel or it is overwritten by a new occurrence of the message, whatever occurs first. Each new instance of a message overwrites the current message when it reaches a destination port, and remains there until it is overwritten. This allows the destination partitions to access the latest message.

A partition's write operation on a specified port is supported by `XM.write_sampling_message()` hypercall. This hypercall copies the message into an internal XtratuM buffer. Partitions can read the message by using `XM.read_sampling_message()` which returns the last message written in the buffer. XtratuM copies the message to the partition space.

Any operation on a sampling port is non-blocking: a source partition can always write into the buffer and the destination partition/s can read the last written message.

The channel has an optional configuration attribute named `@refreshPeriod`. This attribute defines the maximum time that the data written in the channel is considered "valid". Messages older than the valid period are marked as invalid. When a message is read, a bit is set accordingly to the valid state of the message.

Queuing port: It provides support for buffered unicast communication between partitions. Each port has a queue associated where messages are buffered until they are delivered to the destination partition. Messages are delivered in FIFO order.

Sending and receiving messages is performed by two hypercalls: `XM.send_queuing_message()` and `XM.receive_queuing_message()`, respectively. XtratuM implements a classical producer-consumer circular buffer without blocking. The sending operation writes the message from partition space into the circular buffer and the receiving one performs a copy from the XtratuM circular buffer into the destination memory.

If the requested operation cannot be completed because the buffer is full (when trying to send a message) or empty (when attempting to receive a message), then **the operation returns immediately with the corresponding error**. The partition's code is responsible for retrying the operation later.

In order to optimise partition's resources and reduce the performance loss caused by polling the state of the port, XtratuM triggers an extended interrupt when a new message is written/sent to a port. Since there is only one single interrupt line to notify for incoming messages, on the reception of the interrupt, the partition code has to determine which port or ports are ready to perform the operation. XtratuM maintains a bitmap in the Partition Control Table to inform about the state of each port. A "1" in the corresponding entry indicates that the requested operation can be performed.

When a new message is available in the channel, XtratuM triggers an extended interrupt to the destination(s).



2.8.1 Multicore implications

When the plan allocates partitions in several CPU's, partitions involved in a communication can be running or not at the same time. If they do not overlap in time, the reader will receive the interrupt at the beginning of its slot. If they overlap, as soon as the writer partition sends the message, XtratuM will deliver the interrupt the reader partition that will be able to receive the message.

2.9 Health monitor (HM)

The health monitor is the part of XtratuM that detects and reacts to anomalous events or states. The purpose of the HM is to discover errors at an early stage and try to solve or confine the faulting subsystem in order to avoid a failure or reduce the possible consequences.

It is important to clearly understand the difference between 1) an incorrect operation (instruction, function, application, peripheral, etc.) which is handled by the normal control flow of the software, and 2) an incorrect behaviour which affects the normal flow of control in a way not considered by the developer or which can not be handled in the current scope.

An example of the first kind of errors is: the `malloc()` function returns a null pointer because there is not memory enough to attend the request. This error is typically handled by the program by checking the return value. As for the second kind, an attempt to execute an undefined instruction (processor instruction) may not be properly handled by a program that attempted to execute it.

The XtratuM health monitoring system will manage those faults that cannot, or should not, be managed at the *scope* where the fault occurs.

The XtratuM HM system is composed of four logical blocks:

HM event detection:

to detect abnormal states, using logical probes in the XtratuM code.

HM actions:

a set of predefined actions to recover from the fault or confine an error.

HM configuration:

to bind the occurrence of each HM event with the appropriate HM action.

HM notification:

to report the occurrence of the HM events.

Some events are detected directly by the hypervisor. Some hardware events are received directly by the partitions that generate them. As example, FPU faults are addressed directly to the partition who generate them. Aborts caused in the memory access (data or code) are attended in partition environment if are MMU fault and in the hypervisor scope if a TZ fault is released as shown in Figure 2.7.

Partition events must be handled by the partitions and those events are not detected by XtratuM. Therefore, the hypervisor provides to the partition mechanism of notification of events, which will be processed based on actions configured in the configuration file. These services of notification should be used by the partitions in order to maintain the scheme proposed by the hypervisor XtratuM to detect and react to anomalous states.

Since HM events are, by definition, the result of an unexpected behaviour of the system, it may be difficult to clearly determine which is the original cause of the fault, and so, what is the best way to handle the problem. XtratuM provides a set of "coarse grain" actions (see section 2.9.2) that can be used at the first stage, right when the fault is detected. Although XtratuM implements a default action for each HM event, the integrator can map an HM action to each HM event using the XML configuration file.

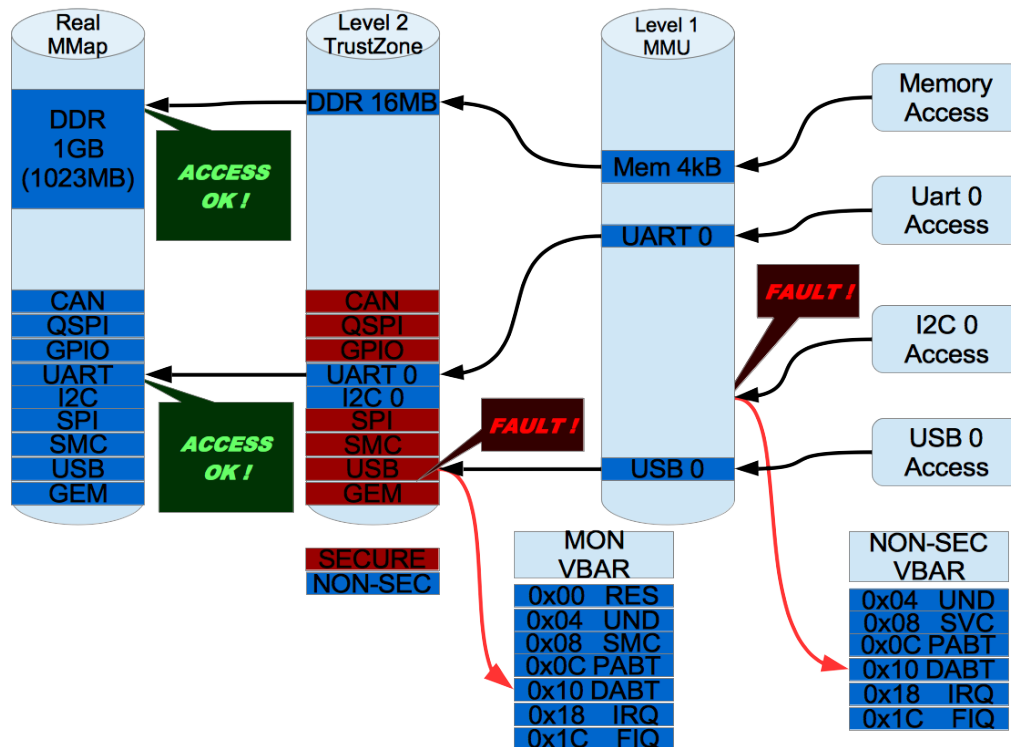


Figure 2.7: Memory access and exceptions.

Once the defined HM action is carried out by XtratuM, a HM notification message is stored in the HM log stream (if the HM event is marked to generate a log). A system partition can then read those log messages and perform a more advanced error handling. As an example of what can be implemented:

450

1. Configure the HM action to stop the faulting partition, and log the event.
2. The system partition can resume an alternate one, a redundant dormant partition, which can be implemented by another developer team to achieve diversity.

Since the differences between *fault*⁴ and *error*⁵ are so subtle and subjective, we will use both terms to refer to the original reason of an incorrect state.

455

The XtratuM health monitoring subsystem defines four different execution scopes, depending on which part of the system has been initially affected:

1. Process scope: Partition process or thread.
2. Partition scope: Partition operating system or run-time support.
3. Hypervisor scope: XtratuM code.
4. Board scope: Resident software (BIOS, BOOT ROM or firmware).

460

The scope⁶ where an HM event should be managed has to be greater than the scope where it is “believed” it can be produced.

⁴Fault: What is believed to be the original reason that caused an error.

⁵Error: The manifestation of a fault.

⁶The term **level** is used in the ARINC-653 standard to refer to this idea

There is not a clear and unique scope for each HM event. Therefore the same HM event may be handled at different scopes. For example, fetching an illegal instruction is considered hypervisor scope if it happens while XtratuM is executing; and partition scope if the event is raised while a partition is running.

XtratuM tries to determine the most likely scope target, and then delivers the HM to the corresponding upper scope.

Note that although in this version of XtratuM there is no distinction between the first and second scopes, it is important to consider that there are two different parts in the partition's code: user applications, and operating system. Therefore, it is consistent to deliver the first scope of HM events, caused by a process or thread, to the second scope.

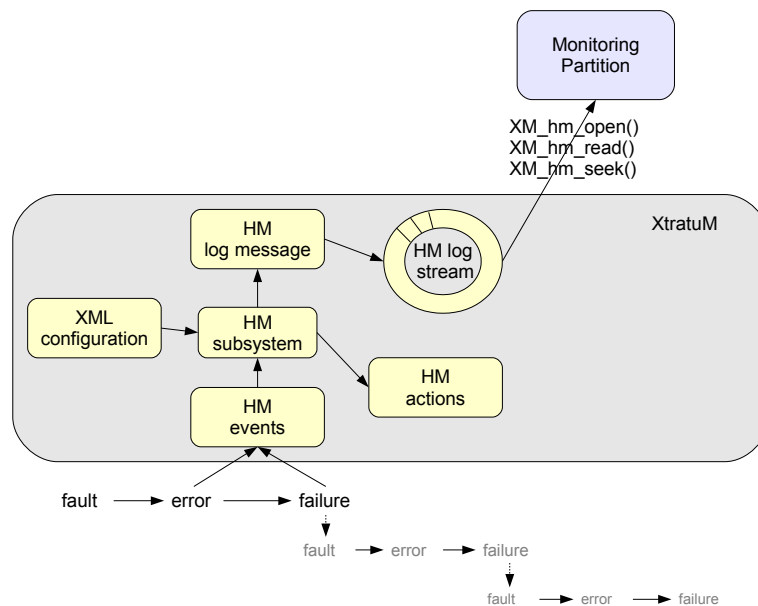


Figure 2.8: Health monitoring overview.

2.9.1 HM Events

There are three sources of HM events:

- Events caused by abnormal hardware behaviour. Most of the processor exceptions are managed as health monitoring events.
- Events detected and triggered by partition code. These events are usually related to checks or assertions on the code of the partitions.
- Events triggered by XtratuM. Caused by a violation of a sanity check performed by XtratuM on its internal state or the state of a partition.

When a HM event is detected, the relevant information (error scope, offending partition identifier, memory address, faulting device, etc.) is gathered and used to select the appropriate HM action.

2.9.2 HM Actions

Once an HM event is raised, XtratuM has to react quickly to the event. The set of configurable HM actions is listed in the next table:

Action	Description
<code>XM_HM_AC_IGNORE</code>	No action is performed.
<code>XM_HM_AC_SHUTDOWN</code>	The shutdown extended interrupt is sent to the failing partition.
<code>XM_HM_AC_COLD_RESET</code>	The failing partition/processor is cold reset.
<code>XM_HM_AC_WARM_RESET</code>	The failing partition/processor is warm reset.
<code>XM_HM_AC_PARTITION_COLD_RESET</code>	The failing partition is cold reset.
<code>XM_HM_AC_PARTITION_WARM_RESET</code>	The failing partition is warm reset.
<code>XM_HM_AC_HYPERVISOR_COLD_RESET</code>	The hypervisor is cold reset.
<code>XM_HM_AC_HYPERVISOR_WARM_RESET</code>	The hypervisor is warm reset.
<code>XM_HM_AC_SUSPEND</code>	The failing partition is suspended.
<code>XM_HM_AC_HALT</code>	The failing partition/processor is halted.
<code>XM_HM_AC_PROPAGATE</code>	No action is performed by XtratuM. The event is redirected to the partition as a virtual trap.
<code>XM_HM_AC_SWITCH_TO_MAINTENANCE</code>	The current scheduling plan is switched to the maintenance one.

485

2.9.3 HM Configuration

There are two tables to bind the HM events with the desired handling actions:

XtratuM HM table: which defines the actions for those events that must be managed at board or hypervisor scope;

Partition HM table: which defines the actions for those events that must be managed at hypervisor or partition scope;

490

Note that the same HM event can be binded with different recovery actions in each partition HM table and in the XtratuM HM table.

The HM system can be configured to send a HM message after the execution of the HM action. It is possible to select whether a HM event is logged or not. See chapter 8.

2.9.4 HM notification

The log events generated by the HM system (those events that are configured to generate a log) are stored in the device configured in the `XM_CF` configuration file.

495

In the case that logs are stored in a log stream, then they can be retrieved by a system partition by using the `XM_hm.X` services.

Health monitoring log messages are fixed length messages defined as follows:

```

struct xmHmLog {
#define XM_HMLOG_SIGNATURE 0xfecf
    xm_u32_t opCode;
#define HMLOG_OPCODE_EVENT_MASK (0x1fff<<HMLOG_OPCODE_EVENT_BIT)
#define HMLOG_OPCODE_EVENT_BIT 19
    // Bits 18 and 17 free

```

500

505

```

510 #define HMLOG_OPCODE_SYS_MASK (0x1<<HMLOG_OPCODE_SYS_BIT)
    #define HMLOG_OPCODE_SYS_BIT 16
    #define HMLOG_OPCODE_VALID_CPUCTX_MASK (0x1<<
        HMLOG_OPCODE_VALID_CPUCTX_BIT)
515 #define HMLOG_OPCODE_VALID_CPUCTX_BIT 15
    #define HMLOG_OPCODE_MODID_MASK (0x7f<<HMLOG_OPCODE_MODID_BIT)
    #define HMLOG_OPCODE_MODID_BIT 8
    #define HMLOG_OPCODE_PARTID_MASK (0xff<<HMLOG_OPCODE_PARTID_BIT)
520 #define HMLOG_OPCODE_PARTID_BIT 0
    xmTime_t timestamp;
    union {
    #define XM_HMLOG_PAYLOAD_LENGTH 4
        struct hmCpuCtxt cpuCtxt;
        xmWord_t payload[XM_HMLOG_PAYLOAD_LENGTH];
    };
    } __PACKED;

525 typedef struct xmHmLog xmHmLog_t;

```

Listing 2.2: core/include/objects/hm.h

signature: A magic number to identify the content of the structure as HM log.

checksum: A MD5 digestion of the data structure allowing to verify the integrity of its content.

opCode: The bits of this field codifies

Bits 31..19 (eventId): Identifies the event that caused this log.

Bits 16 (sys): Set if the HM event was generated by the hypervisor, cleared otherwise.

Bits 15..8 (validCpuCtxt): Set if the field cpuCtxt (see below) holds a valid processor context.

Bits 7..0 (partitionId): The Id attribute of the partition that caused the event.

seq: Number of sequence enabling to sort the event respect other events. It is codified as unsigned integer, incremented each time a new HM event is logged.

timeStamp: A time stamp indicating when the event was detected.

Either cpuCtxt or payload: When validCpuCtxt bit is set (opCode), then this field holds the processor context when the HM event was generated, otherwise, this field may hold information of the generated event. For instance, an application can manually generate a HM event specifying this information.

2.9.5 Multicore implications

Health monitor management has not visibility on the VCPUs internal to the partitions. From this point of view, a trap generated by a VCPU will:

- will raise a HM event
- XtratuM will handle the event as execute the action defined in the configuration file which is common to all the internal VCPUs.
- In the case of a propagation action, XtratuM identifies the VCPU that generated the trap and delivers to the causing VCPU the associated virtual interrupt
- In the case of a partition action (halt, reset, etc.), the action is applied independently of the causing VCPU.

2.10 Access to devices

The partition is in charge of handling properly the device. No driver is provided by XtratuM. Integrator is in charge to assign the device in the proper way for each partition.

550

Devices could be accessed in two different ways:

- Mapping the device as a memory area and accessing it directly.
- Creating one or more IOPort(s) that allows the partition access the device.

Section 5.8 provide an extensive explanation of how to configure each of them.

555

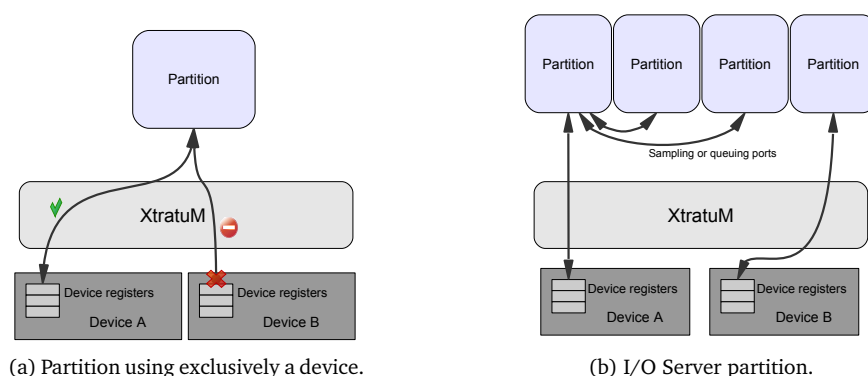


Figure 2.9: Device Access Models.

Two partitions can not use neither the same device nor a interrupt line. When a device is used by several partitions, a user implemented I/O server partition (figure 2.9b) may be in charge of the device management. An I/O server partition is a specific partition which accesses and controls the devices attached to it, and exports a set of services via the inter-partition communication mechanisms provided by XtratuM (sampling or queuing ports), enabling the rest of partitions to make use of the managed peripherals. The policy access (priority, FIFO, etc.) is implemented by the I/O server partition. Additionally other partitions can access devices that have been assigned exclusively.

560

Note that the I/O server partition is not part of XtratuM. It should, if any, be implemented by the user of XtratuM. XtratuM does not force any specific policy to implement I/O server partitions but a set of services to implement it.

565

2.11 Exceptions

An **exception** causes the processor to suspend program execution to handle an event, such as an externally generated interrupt or an attempt to execute an undefined instruction. Exceptions can be generated by internal or external sources.

In ARM CORTEX-A9 architecture, the exception vectors are eight consecutive word-aligned memory addresses, starting at an exception base address. Exist 2 different exception vectors, one is used as regular exception vector while the Monitor Vector is used to handle the Secure Monitor Call (SMC) and those exceptions configured to be attended in monitor mode.

570

The exception base address is set in the register VBAR and MVBAR for regular and Monitor vectors respectively. VBAR is banked between Secure and Non-Secure world, it is hypervisor and partitions. Therefore, in summary there are two exception vectors but one is duplicated:

575

- Regular(Secure - hypervisor),
- Regular(Non-Secure - partitions),
- Monitor (hypervisor).

Table 2.4 contains the offset for each exception with respect to the exception base address.

Offset	Secure Exceptions VBAR	Non-Secure Exceptions NS-VBAR	Monitor Exceptions MVBAR
0x00	Reset	Reset	<i>Not used</i>
0x04	Undefined Instruction	Undefined Instruction	<i>Not used</i>
0x08	Supervisor Call (SVC)	Supervisor Call (SVC)	Secure Monitor Call (SMC)
0x0C	Prefetch Abort	Prefetch Abort	Prefetch Abort
0x10	Data Abort	Data Abort	Data Abort
0x14	<i>Not used</i>	<i>Not used</i>	<i>Not used</i>
0x18	Interrupt (IRQ)	Interrupt (IRQ)	Interrupt (IRQ)
0x1C	Fast Interrupt (FIQ)	Fast Interrupt (FIQ)	Fast Interrupt (FIQ)

Table 2.4: Exception offsets for ARM CORTEX-A9.



For each partition, XtratuM defines the exception vector (NS-VBAR) in the same address of the entryPoint of the partition. Usually this is placed at the beginning of partition code memory map.

Most exceptions are not emulated by XtratuM. These are generated by hardware and delivered directly to the partition. The **Reset** exception is emulated at the beginning of the partition execution and at any Reset exception occurrence.

An **Extended IRQ** is an exception generated by XtratuM to inform the partition about specific events. Each of these exception are emulated when unmasked and interrupts are enabled, otherwise, it is left as pending.

For more information about the mechanism used by XtratuM to distinguish between them, and about the convention to be followed to request para-virtualised services, please refer to section 6.2.

2.11.1 Interpartition virtual interrupt management

Additionally to the interrupts propagated by XtratuM to the partitions, XtratuM provides the mechanisms to send an interrupt from a partition to other or others. This are the inter-partition virtual interrupts (IPVI). A IPVI is an interrupt, or more properly an event, generated by a partition to inform to other or others partitions about some relevant state.

Next table shows the list of IPVIs available to be used by the partitions.

```

.....
/* Inter-Partition Virtual Interrupts */
#define XM_MAX_IPVI          CONFIG_XM_MAX_IPVI
#define XM_VT_EXT_IPVI0      (24+XM_VT_EXT_FIRST)
#define XM_VT_EXT_IPVI1      (25+XM_VT_EXT_FIRST)
#define XM_VT_EXT_IPVI2      (26+XM_VT_EXT_FIRST)
#define XM_VT_EXT_IPVI3      (27+XM_VT_EXT_FIRST)
#define XM_VT_EXT_IPVI4      (28+XM_VT_EXT_FIRST)
#define XM_VT_EXT_IPVI5      (29+XM_VT_EXT_FIRST)
#define XM_VT_EXT_IPVI6      (30+XM_VT_EXT_FIRST)
#define XM_VT_EXT_IPVI7      (31+XM_VT_EXT_FIRST)

```

Listing 2.3: sources/core/include/guest.h

IPVIs are raised by partitions without the explicit knowledge of the partition or partitions that will receive them. The connection or channel that defines the IPVI route is specified in the configuration file (XM.CF). Next example shows the specification of the IPVI route.

```
<Channels>
  <Ipvi id="0" sourceId="0" destinationId="1 2" />
  <Ipvi id="1" sourceId="0" destinationId="2" />
</Channels>
```

Listing 2.4: IPVI route specification

It defines that XM_VT_EXT_IPVIO() (Ipvi identifier 0) can be generated by partition with identifier 0 and it will be received by partitions with identifier 1 and 2. Additionally, the same partition 0 can raise XM_VT_EXT_IPVI1() (Ipvi with identifier 1) that can be received by partition 2.

2.11.2 Multicore implications

Interrupts are allocated to the partitions in the configuration file. In multicore operating system, it is responsible of attach the interrupts to the CPUs. In the same way, the guest OS is responsible of the assignment of the interrupts allocated to it, to the appropriated VCPU.

2.12 Traces

XtratuM provides a mechanism to store and retrieve the traces generated by partitions and XtratuM itself. Traces can be used for debugging, during the development phase of the application, but also to log relevant events during the production phase.

In order to enforce resource isolation, each partition (as well as XtratuM) has a dedicated trace log stream to store its own trace messages. The log stream mechanism is implemented as circular buffer which is able to store a maximum of CONFIG_OBJ.TRACE.LOG.NO_ELEM elements. When it gets full, the generation of a new trace event overwrites the oldest stored trace event.

System partitions can retrieve the stored trace events by using the XM_read_trace() hypercall. This hypercall is destructive, that is, once a trace is read, it is removed from the log stream.

2.12.1 Multicore implications

No implications.

2.13 Clocks and timers

There are two clocks per partition. Both are virtual and provided and managed by XtratuM.

XM_HW_CLOCK: Associated with the native hardware clock. The resolution is 1μsec.

XM_EXEC_CLOCK: Associated with the execution of the partition. This clock only advances while the partition is being executed. It can be used by the partition to detect overruns. This clock relies on the XM_HW_CLOCK and its resolution is also 1μsec.

Only one timer can be armed for each clock.

In the multicore approach, XtratuM provides:

- one global hardware clock (XM_HW_CLOCK) global for all *virtual CPU*
- one local execution clock (XM_EXEC_CLOCK) for each *virtual CPU*

645 Each *virtual CPU* can control its execution time using the local execution clock.

Associated to each clock, the partition can arm one timer. In the multicore approach, the partition (any of the *virtual CPU*) can arm one time based on the global hardware clock. Each *virtual CPU* can arm a one timer based on the local execution clock.

650 It is responsibility of the guestOS or partition runtime of handling more than one timers based on the same clock using the appropriated data structures.

2.14 Symmetric Multi-Processing (SMP) architecture

A SMP architecture is a multiprocessor hardware where two or more indential processors are connected to a single shared main memory, having full access to all I/O devices.

655 In the case of XtratuM and SMP architectures, a single instance of XtratuM manages all the processors on the board equally but during the initialisation stage. Spinlocks are used to protect race conditions within XtratuM.

In this architecture, the primary processor starts up the system, as in the mono-core version. Once this sequence is finished, this processor initialises secondary processors and waits in a synchronisation point to start partition's scheduling.

660 Scheduler plan definition has been extended to include additional processors. Each processor defines its own plan or set of plans used by the scheduler to decide *who* has to execute *what*.

Moreover, in order to better control the hypervisor's output, a new XML configuration attribute **@console** has been added to the HwDescription/ProcessorTable/Processor element. The use of this attribute enables that all the output produced by this processor when in supervisor mode is redirected to that console.

665 Multi-core partitions are also supported by introducing the concept of *virtual CPU* (vCPU). A vCPU in a partition is seen as a hardware processor if no virtualisation layer were present. A partition has as many vCPUs as configured during the configuration of XtratuM (this number could not match with the number of real processors). For compatibility reasons, previous versions of XtratuM are considered as partitions with just one vCPU.

670 As in the real hardware, XtratuM just starts up the first vCPU (vCPU0), the rest are left halted. The partition itself is in charge of waking up as many vCPUs as required.

The following hypercalls have been added in order to manage vCPUs:

- `XM_reset_vcpu()`: resets a vcpu from the calling partition. This hypercalls requires as arguments the ID of the vCPU to be resetted, the entry point where the execution is started and a value that is passed to the vCPU.
- `XM_halt_vcpu()`: halts a vCPU. It requires the ID of the vCPU to be halted.
- `XM_get_vcpuid()`: returns the ID of the calling vCPU.

Moreover, the partition control table has been replicated as many times as vCPUs are present in the partition. Each vCPU shall use the entry which corresponds to its ID.

The XML has been also updated to assume the presence of additional processors and the vCPUs: The set of plans has been replicated as many times as processors are on the board, one set for each processor. At execution time, each processor uses its own set of plans. Besides, each scheduling plan's slot includes the ID of the vCPU to be scheduled.

The following example shows a board where two processors are present. Each processor defines only one plan. Processor 0 executes the vCPU 0 and 1 of the partition 5 and 20, respectively, whereas Processor 1 executes the vCPU 1 and 0 of the same partitions. In this case, both processors shall execute parallelly in both partitions.

```
...
<ProcessorTable>
  <Processor id="0" frequency="400Mhz">
    <CyclicPlanTable>
      <Plan id="0" majorFrame="2000ms">
        <Slot id="0" start="0ms" duration="1000ms" partitionId="5" vCpuId="0" />
        <Slot id="1" start="1100ms" duration="200ms" partitionId="20" vCpuId="1" />
      </Plan>
    </CyclicPlanTable>
  </Processor>
  <Processor id="1" frequency="400Mhz">
    <CyclicPlanTable>
      <Plan id="0" majorFrame="2000ms">
        <Slot id="0" start="0ms" duration="1000ms" partitionId="5" vCpuId="1" />
        <Slot id="1" start="1100ms" duration="200ms" partitionId="20" vCpuId="0" />
      </Plan>
    </CyclicPlanTable>
  </Processor>
</ProcessorTable>
...
```

2.14.1 Snoop Control Unit (SCU)

The SCU block maintains the data cache coherence between the individual L1 data cache in the two ARM CORTEX-A9 processors and the L2 cache.

The SCU is enabled by default in XtratuM. This feature is configured to support parity error detection (see 5.9.2).

2.14.2 Branch Prediction

The ARM CORTEX-A9 processor implements both static and dynamic branch prediction. The dynamic branch prediction logic employs a global branch history buffer (GHB) table which is updated every time a branch gets executed. The branch execution and the overall instruction throughput also benefit greatly from the implementation of a branch target address cache (BTAC) which holds the target addresses of the recent branches.

The branch prediction is enabled by default in the hypervisor. The system is configured to detect parity errors related to GHB data or BTAC data, which are automatically corrected when the branch gets executed (see 5.9.2).

2.15 Inter-Partition Virtual Interrupt (IPVI)

An Inter-Partition Virtual Interrupt (IPVI) emulates the way a real Inter-Processor Interrupts (IPIs) works in real processors. That is, every time the correspondent hypercall is invoked, a virtual interrupt

is caused to a destination partition. An IPVI can be raised by any partition.

Each partition has a maximum of 8 IPVIs, implemented as the last eight extended virtual interrupts. The system integrator, though the XML, indicates the entity who receives a IPVI after being raised.

```

725 <Channels>
    <Ipvi id="0" sourceId="5" destinationId="8" />
    <Ipvi id="0" sourceId="4" destinationId="1" />
    <Ipvi id="1" sourceId="4" destinationId="8, 1" />
</Channels>

```

730 The example above shows a configuration where the behaviour of three IPVIs is defined: the IPVI 0, caused by the partition 5 and received by the partition 8. The IPVI 0 and 1, caused both by the partition 4 and received, the first one by the partition 1 and the second by the partitions 8 and 1. The hypercall `XM.raise_ipvi()` has been included in order to enable partitions to cause IPVIs.

2.16 Status

Relevant internal information regarding the current state of the XtratuM and the partitions, as well as accounting information is maintained in an internal data structure that can be read by system partitions.



This optional feature shall be enabled in the XtratuM source configuration, and then recompile the XtratuM code. **By default it is disabled.** The hypercall is always present; but if not enabled, then XtratuM does not gather statistical information and then some status information fields are undefined. It is enabled in the XtratuM menuconfig: Objects → XM partition status accounting.

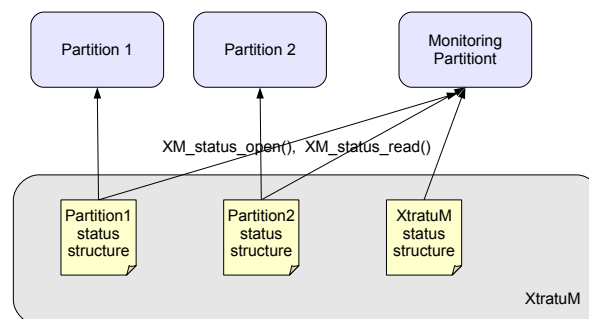


Figure 2.10: Status overview.

2.17 Summary

740 Next, there is a brief summary of the ideas and concepts that shall be kept in mind to understand the internal operation of XtratuM and how to use the hypercalls:

- A partition behaves basically as the native computer. Only those services that have been explicitly para-virtualised should be managed in a different way.
- Partition's code may not be self-modifying. The partition is responsible to appropriately flush cache in order to guarantee the coherency.
- Partition's code is always executed with native fast interrupts enabled. This behaviour is enforced by XtratuM. native interrupts could be configured by the partition.

- Partition's code is only allowed to manage their own native interrupts and their virtual interrupts.
- XtratuM code is non-preemptive. It should be considered as a single critical section.
- Partitions are scheduled by using a predefined scheduling cyclic plan.
- Inter-partition communication is done through messages.
- There are two kinds of virtual communication devices: sampling ports and queuing ports.
- All hypercall services are non-blocking.
- Regarding the capabilities of the partitions, XtratuM defines two kinds of partitions: system and standard.
- Only system partitions are allowed to control and to query the state of the system and of other partitions.
- XtratuM is configured off-line and no dynamic objects can be added at run-time.
- The XtratuM configuration file (XM_Cf) describes the resources that are allowed to be used by each partition.
- XtratuM provides a fine grain error detection and a coarse grain fault management.
- It is possible to implement advanced fault analysis techniques in system partitions.
- An I/O Server partition can handle a set of devices used by several partitions.
- XtratuM implements a highly configurable health monitoring and handling system.
- The logs reported by the health monitoring system can be retrieved and analysed by a system partition online.
- XtratuM provides a tracing service that can be used to both debug partitions and online monitoring.
- The same tracing mechanism is used to handle partition and XtratuM traces.



750

755

760

765

This page is intentionally left blank.

Chapter 3

Development Process Overview

XtratuM is a layer of software that extends the capabilities of the native hardware. There are important differences between a classical system and a hypervisor based one. This chapter provides an overview of the XtratuM developing environment. 770

The simplest scenario is composed of two actors: the *System Integrator* and the *Partition Developer* or partition supplier. There shall be only one integrator team and one or more partition developer teams (in what follows, we will use “integrator” and “partition developer” for short). 775

The tasks to be done by the **integrator** are:

1. Configure the XtratuM source code (jointly with the resident software). Customise it for the target board (processor model, etc.) and a miscellaneous set of code options and limits (debugging, identifiers length, etc.). See section 8.1 for a detailed description.
2. Build XtratuM: hypervisor binary, user libraries and tools. 780
3. Distribute the resulting binaries to the partition developers. All partition developers shall use the same binary version of XtratuM.
4. Allocate the available system resources to the partitions, according to the resources required to execute each partition:
 - memory areas where each partition will be executed or can use, 785
 - design the scheduling plan,
 - communication ports between partitions,
 - the virtual devices and physical peripherals allocated to each partition,
 - configure the health monitoring,
 - etc. 790

By creating the XM_CF configuration file¹. See section 8.2 for a detailed description.

5. Gather the partition images and customisation files from partition developers.
6. Pack all the files (resident software, XtratuM binary, partitions, and configuration files) into the final system image.

The **partition developer** activity: 795

¹Although it is not mandatory to name “XM_CF” the configuration file, we will use this name in what follows for simplicity.

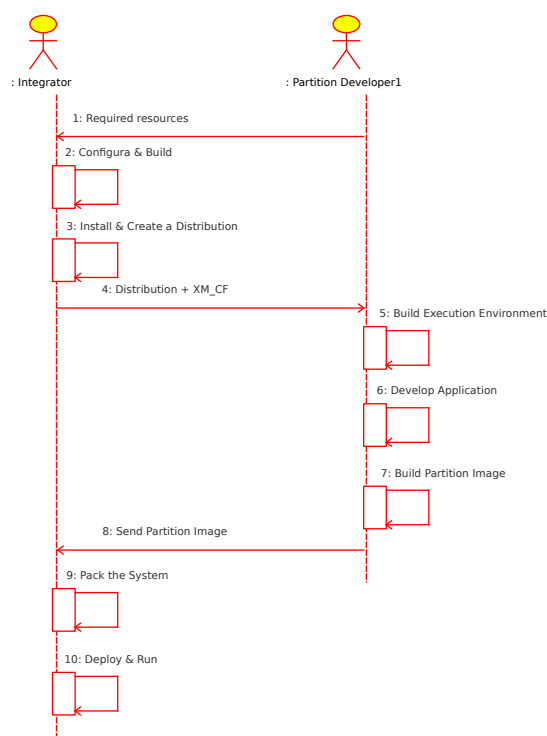


Figure 3.1: Integrator and partition developer interactions.

1. Define the resources required by its application, and send it to the integrator.
2. Prepare the development environment. Install the binary distribution created by the integrator.
3. Develop the partition application, according to the system resources agreed by the integrator.
4. Deliver to the integrator the resulting partition image and the required customisation files (if any).

There should be an agreement between the integrator and the partition developers on the resources allocated to each partition. The binaries, jointly with the XM_CF configuration file defines the partitioned system. **All partition developers shall use exactly the same XtratuM binaries and configuration files during the development.** Any change on the configuration shall be agreed with the integrator.

Since the development of the partitions may be carried out in parallel (or due to intellectual property restrictions), the binary image of some partitions may not be available to a partition developer team. In this case, it is advisable to use dummy partitions to replace those non-available, rather than changing the configuration file.

3.1 Development at a glance

- ① The first step is to build the hypervisor binaries. The integrator shall configure and compile the XtratuM sources to produce:

xm_core.xef: The hypervisor image which implements the support for partition execution.

libxm.a: A helper library which provides a “C” interface to the para-virtualised services via the hypercall mechanism.

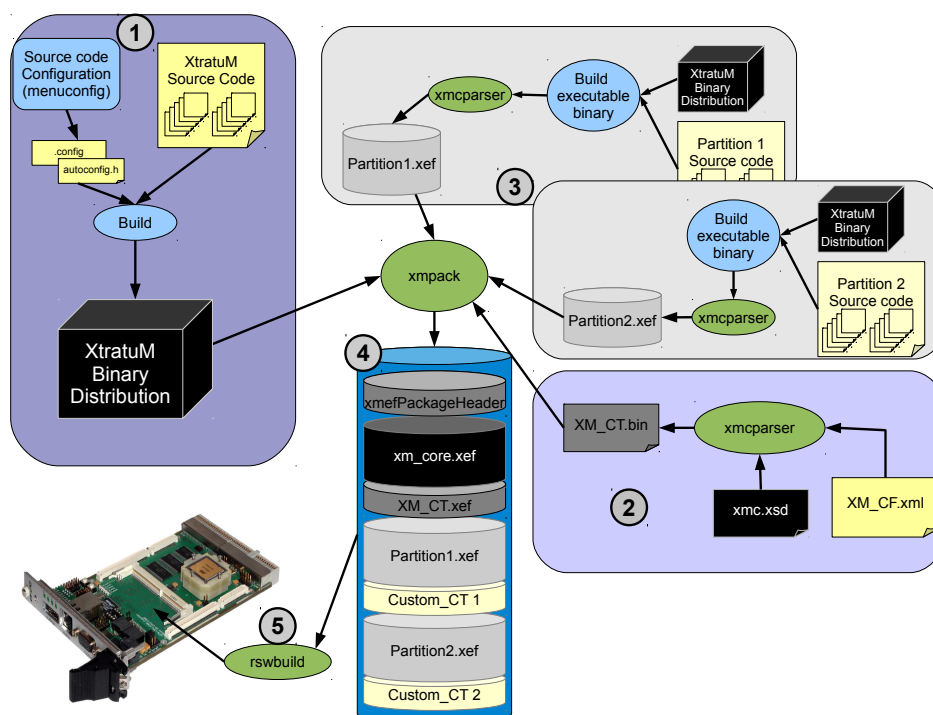


Figure 3.2: The big picture of building a XtratuM system.

xmc.xsd: The XML schema specification to be used in the XM_CF configuration file.

tools: A set of tools to manage the partition images and the XM_CF file.

815

The result of the build process can be prepared to be delivered to the partition developers as a binary distribution.

- ② The next step is to define the hypervisor system and resources allocated to each partition. This is done by creating the configuration file XM_CF file.
- ③ Using the binaries resulted from the compilation of XtratuM and the system configuration file, partition developers can implement and test its own partition code by their own.
- ④ The tool xmpack is used to build the complete system (hypervisor plus partitions code). The result is a single file called *container*. Partition developers shall replace the image of non-available partitions by a dummy partition. Up to a maximum of CONFIG_MAX_NO_CUSTOMFILES customisation files can be attached to each partition.
- ⑤ The container shall be loaded in the target system using the corresponding resident software (or boot loader). For convenience, a resident software is provided.

820

825

3.2 Building XtratuM

In the first stage, **XtratuM shall be tailored to the hardware available on the board, and the expected workload.** This configuration parameters will be used in the compilation of the XtratuM code

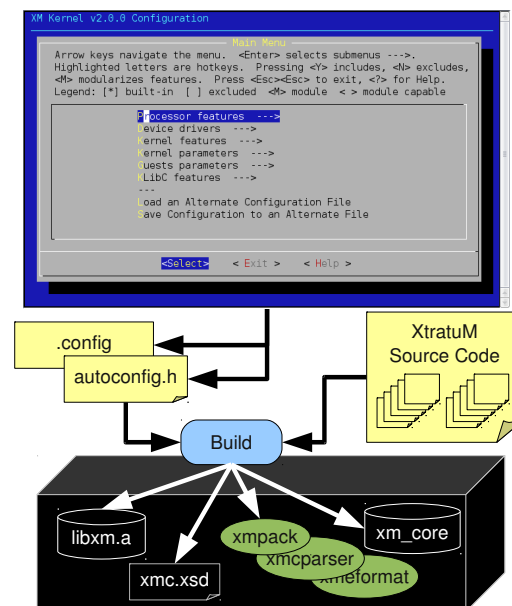


Figure 3.3: Menuconfig process.

to produce a compact and efficient XtratuM executable image. Parameters like the processor model or the memory layout of the board are configured here (see section 8.1).

The configuration interface is the same as the one known as “*menuconfig*” used in the Linux kernel, see figure 3.3. It is a ncurses-based graphic interface to edit the configuration options. The selected choices are stored in two files: a “C” include file named “core/include/autoconf.h”; and a Makefile include file named “core/.config”. Both files contain the same information but with different syntax in order to be used in “C” programs and in Makefiles, respectively.

Although it is possible to edit these configuration files, with a plain text editor, it is advisable not to do so; since both files shall be synchronized.

Once configured, the next step is to build XtratuM binaries, which is done calling the command `make`.

Ideally, configuring and compiling XtratuM should be done at the initial phases of the design and should not be changed later.

The build process leaves the objects and executables files in the source directory. Although it is possible to use these files directly to develop partitions it is advisable to install the binaries in a separate read-only directory to avoid accidental modifications of the code. It is also possible to build a TGZ² package with all the files to develop with XtratuM, which can be delivered to the partition developers. See chapter 4.

3.3 System configuration

The integrator, in agreement with the partition developers, have to define the resources allocated to each partition, by creating the `XM_CF` file. It is an XML file which shall be a valid XML against the XMLSchema defined in section 8.2. Figure 3.4 shows a graphical view of the configuration schema.

The main information contained in the `XM_CF` file is:

²TGZ: Tar Gzipped archive.

Memory: The amount of physical memory available in the board and the memory allocated to each partition.

Processor: How the processor is allocated to each partition: the scheduling plan.

Peripherals: Those peripherals not managed by XtratuM can be used by one partition. The I/O port ranges and the interrupt line if any.

855

Health monitoring: How the detected errors are managed by the partition and XtratuM: direct action, delivered to the offending partition, create a log entry, reset, etc.

Inter-partition communication: The ports that each partition can use and the channels that link the source and destination ports.

Tracing: Where to store trace messages and what messages shall be traced.

860

Since XM_CF defines the resources allocated to each partition, this file represents a **contract between the integrator and the partition developers**. A partner (the integrator or any of the partition developers) should not change the contents of the configuration file on its own. All the partners should be aware of the changes and should agree in the new configuration in order to avoid problems later during the integration phase.



865

In order to reduce the complexity of the XtratuM hypervisor, the XM_CF is parsed and translated into a binary format which can be directly used by XtratuM. The XML data is translated into a set of initialised data structures ready to be used by XtratuM. Otherwise, XtratuM would need to contain a XML parser to read the XM_CF information. See section 9.1.1.

The resulting configuration binary is passed to XtratuM as a “customisation” file.

870

3.4 Compiling partition code

Partition developers should use the XtratuM user library (named `libxm.a` which has been generated during the compilation of the XtratuM source code) to access the para-virtualised services. The resulting binary image of the partition shall be self-contained, that is, it shall not contain linking information. The ABI of the partition binary is described in section 6.

In order to be able to run the partition application, each partition developer requires the following files:

875

libxm.a: Para-virtualised services. The include files are distributed jointly with the library, and they should be provided by the integrator.

XM_CF.xml: The system configuration file. This file describes the whole system. The same file should be used by all the partners.

880

xm_core.bin: The hypervisor executable. This file is also produced by the integrator, and delivered to the other partners.

xmpack: The tool that packs together, into a single system image container, all the *components*.

xmeformat: Tool used to translate an ELF file into a XEF one.

xmcparser: The tool to translate the configuration file (XM_CF.xml) into a “C” file which could be compiled to produce the configuration table (XM_CT).

885

Partition developer should use an execution environment as close as possible to the final system: the same processor board and the same hypervisor framework. To achieve this goal, they should use the same configuration file as the one used by the integrator. But the code of other partitions may be replaced by dummy partitions. This dummy partition code could execute, for instance, just a busy loop to waste time.

890

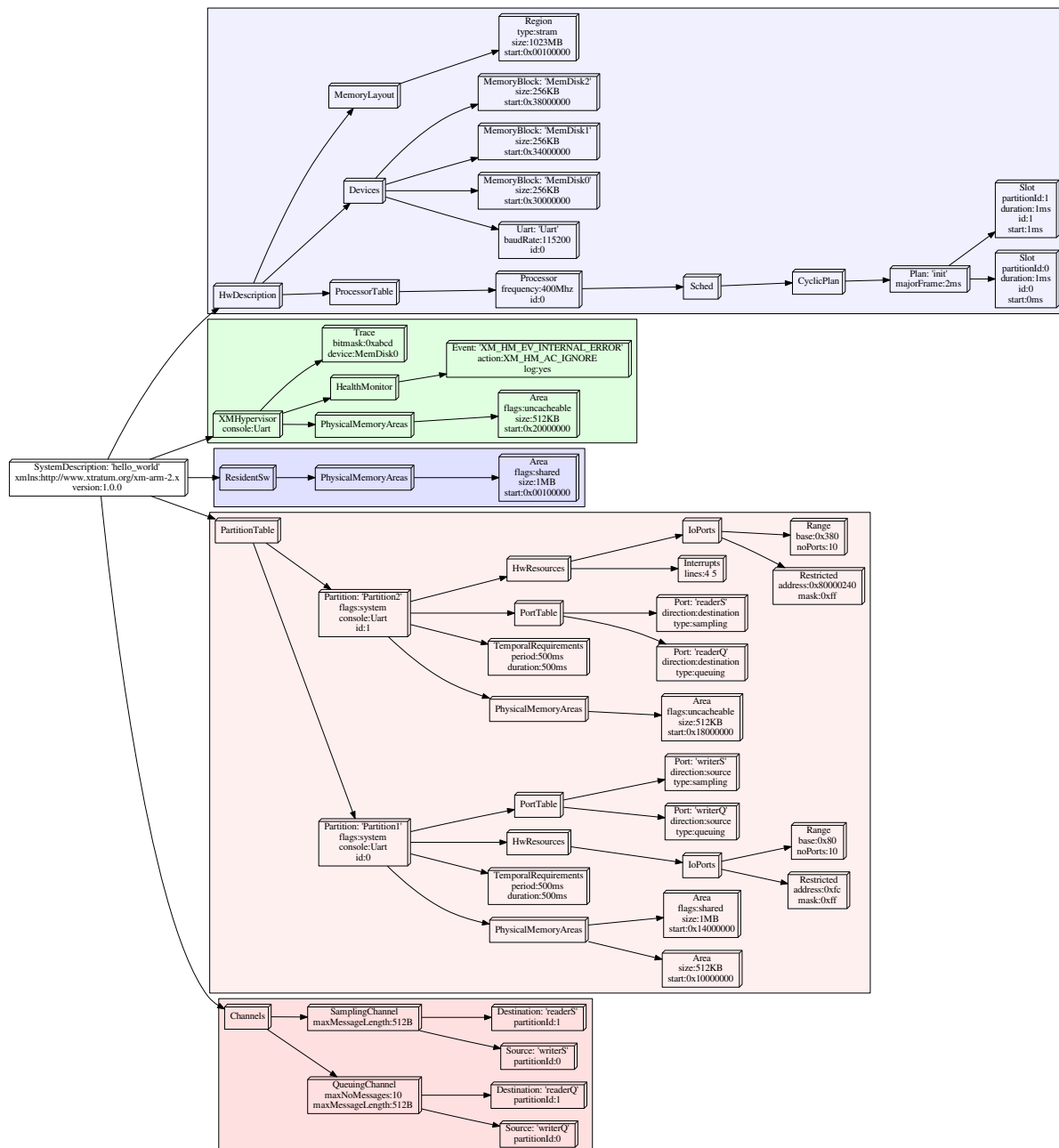


Figure 3.4: Graphical representation of an XML configuration file.

3.5 Passing parameters to the partitions: customisation files

User data can be passed to each partition at boot time. This information is passed to the partition via the *customisation* files.

It is possible to attach up to a maximum of `CONFIG.MAX_NO_CUSTOMFILES` customisation files per partition. The content of each customisation file is copied into the partition memory space at boot time (before the partition boots). The buffer where each customisation file is loaded is specified in the partition header by the partition developer. See section 6.

895

This is the mechanism used by XtratuM to get the compiled XML system configuration.

3.6 Building the final system image

The partition binary is not an ELF file. It is a custom format file (called *XEF*) which contains the machine code and the initialized data. See section 6.

900

The *container* is a **single file** which contains all the code, data and configuration information that is loaded in the target board. In the context of the container, a *component* refers to the set of files that are part of an execution unit (which can be a partition or the hypervisor itself). `xmpack` is a program that reads all the executable images (XEF files) and the configuration/customisation files and produces the container.

905

The container is not a bootable code. That is, it is like a “tar” file which contains a set of files. In order to be able to start the partitioned system, a boot loader shall load the content of the container into the corresponding partition addresses. The utility `rsbuild` creates an bootable ELF file with the resident software and the container.

This page is intentionally left blank.

Chapter 4

Building XtratuM

4.1 Development environment

XtratuM has been compiled and tested with the following development tools:

910

Package	Version	Linux package name		Purpose
host gcc	4.6.8/4.8.2	gcc-4.6/gcc-4.8	req	Build host utilities
host binutils	2.24	binutils	req	Build host utilities
make	3.81-8	make	req	Core Building
makeself	2.2.0	makeself	req	Build self extracting distro
libncurses	5.9.20140118	libncurses5-dev	req	Configure source code
libxml2	2.7.8/2.9.1	libxml2-dev	req	Configure XtratuM parser
arm-xilinx-eabi-gcc	4.8.3	GCC(ARM Xilinx CC)	req	Core Building
arm-xilinx-eabi-ld	2.24.51	GNU Binutils (ARM Xilinx CC)	req	Core Building
arm-xilinx-eabi-objcopy	2.24.51	GNU Binutils (ARM Xilinx CC)	req	Core Building
arm-xilinx-eabi-as	2.24.51	GNU Binutils (ARM Xilinx CC)	req	Core Building
arm-xilinx-eabi-ar	2.24.51	GNU Binutils (ARM Xilinx CC)	req	Core Building
xmllint	2.7.8/2.9.1	libxml2-utils	req	Configure XtratuM parser
python	2.7.6	python	opt	Tools
perl	5.18.2	perl	opt	Testing
Xilinx SDK	2014.2/2014.3	Tools Xilinx SDK	req	Manage board connection, ARM Xilinx toolchain
uBoot	2011.03-dirty	uBoot provided by Xilinx	req	Board bootloader

Packages marked as “req” are required to compile XtratuM. Those packages marked as “opt” are needed to compile or use it in some cases.

4.2 Compile XtratuM Hypervisor

It is not required to be supervisor (root) to compile and run XtratuM.

915

The first step is to prepare the system to compile XtratuM hypervisor.

1. Check that the GNU LIBC Linux GCC 4.8.3 toolchain for ARM is installed in the system. It can be obtained from Xilinx.
2. Make a deep clean to be sure that there is not previous configurations:

```
$ make distclean
```

3. In the root directory of XtratuM, copy the file `xmconfig.arm` into `xmconfig`, and edit it to meet your system paths. The variable `XTRATUM_PATH` shall contain the root directory of XtratuM. Also, if the ARM CORTEX-A9 cross-compiler toolchain directory is not in the `PATH` then the variable `TARGET_CCPREFIX` shall contain the path to the actual location of the corresponding tools.

In the seldom case that the host toolchain is not in the `PATH`, then it shall be specified in the `HOST_CCPREFIX` variable.

```
$ cp xmconfig.arm xmconfig
$ vi xmconfig .....
```

4. Configure the XtratuM sources. The `ncurses5` library is required to compile the configuration tool. In a Debian system with internet connection, the required library can be installed with the following command: `sudo apt-get install libncurses5-dev`.

The configuration utility is executed (compiled and executed) with the next command:

```
$ make menuconfig
```

Note: When the `menuconfig` target is launched, three different configuration menus are presented:

- (a) The XtratuM itself.
- (b) The Resident Software, which is charge of loading the system from ROM to RAM.
- (c) The XAL, a basic partition execution environment.

Section 8.1 contains more information about the available configuration options and their purpose.

Additionally, the user can use a configuration defined by default for the board. The default configuration can be used through the command `make defconfig`.

```
$ make defconfig
> Building Kconfig:
> Target architecture: [arm]
```

Listing 4.1: Loading configurations by default

This option of the `Makefile` configures XtratuM, XAL and RSW to be used with the board. Memory map of the `xml` files in the XAL examples is prepared for this default configuration.

The User could save their own configuration files. This files must be placed next to the `defconfig` files. The name of this file must follow the pattern `defconfig-<user name>`.

This user configuration files could be load through the command `make defconfig-<user name>`

```
$ make defconfig-myconfig
> Building Kconfig:
> Target architecture: [arm]
```

Listing 4.2: Loading user defined configurations "defconfig-myconfig"

5. Compile XtratuM sources:

```

$ make
> Configuring and building the "XtratuM hypervisor"
> Building XM Core
  - kernel/arm
  - kernel/mmu
  - kernel
  - klibc
  - klibc/arm
  - objects
  - drivers
> Linking XM Core
   text  data  bss   dec   hex filename
110078   256  32764 143098 22efa xm_core
aabdae154ab8699ef2335af0b7640da9 xm_core.xef
> XM Core assign attributes
> Done

> Configuring and building the "User utilities"

> Building XM user
  - libxm
  - bootloaders/rsw
  - tools
  - tools/xmpack
  - tools/xmcparser
  - tools/xmbuildinfo
  - tools/rswbuild
  - tools/xef
  - tools/elfbdr
  - xal
> Done

```

4.3 Generating a binary distribution

When the compilation of XtratuM finished, the files required to build a whole XtratuM-based system lay in different places of the source code directory tree. In order to share them with the partition developers a distribution package shall be generated. There are two distribution formats:

Tar file: It is a compressed tar file with all the XtratuM files and an installation script.

```
$ make distro-tar
```

Self-extracting installer: It is a single executable file which contains the distribution and the installation script.

```
$ make distro-run
```

The final installation is exactly the same regarding the distribution format used.

```

$ make distro-run
.....
> Installing XM in "/tmp/xtratum-2.0.0-53884/xtratum-2.0.0/xm"
  - Generating XM sha1sums
  - Installing XAL
  - Generating XAL sha1sums
  - Installing XM examples
  - Generating XM examples sha1sums
  - Setting read-only (og-w) permission.
> Done

> Generating XM distribution "xtratum-2.0.0.tar.bz2"
> Done

> Generating self extracting binary distribution "xtratum-2.0.0.run"
> Done

```

960 The files `xtratum-2.0.0.tar.bz2` or `xtratum-2.0.0.run` contain all the files requires to work (develop and run) with the partitioned system. This tar file contains two root directories: `xal` and `xm`, and an installation script.

```

INSTALL_PATH
|-- xal
|   |-- bin
|   |-- common
|   |-- include
|   |   \-- arch
|   \-- lib
|-- xal-examples
|   |-- example.001
|   |-- example.002
|   |-- example.003
|   |-- example.004
|   |-- example.005
|   |-- example.006
|   |-- example.007
|   \-- hello_world
\-- xm
    |-- bin
    |-- include
    |   |-- arch
    |   \-- xm_inc
    |       |-- arch
    |       \-- objects
    \-- lib

```

Listing 4.3: Installation tree.

965 The directory `xm` contains the XtratuM kernel and the associated developer utilities. `Xal` stands for *XtratuM Abstraction Layer*, and contains the partition code to setup a basic “C” execution environment. `Xal` is provided for convenience, and it is not mandatory to use it. `Xal` is only useful for those partitions with no operating system.



Although XtratuM core and related libraries are compiled for the ARM CORTEX-A9 processor, some of the host configuration and deploying tools (`xmcparser`, `xmpack` and `xmeformat`) are host executables. If the computer where XtratuM was compiled on and the computer where it is being installed are different processor architectures (32bit and 64bit), the tools may not run properly.

970

4.4 Installing a binary distribution

Decompress the `xtratum-2.0.0.tar.bz2` file in a temporal directory, and execute the install script. Alternatively, if the distributed file is `xtratum-2.0.0.run` then just execute it.

The install script requires only two parameters:

1. The installation path.
2. The path to the cross-compile toolchain.

975

Note that it is assumed that the host toolchain binaries can be located in the `PATH` variable. It is necessary to provide again the path to the ARM CORTEX-A9 toolchain because it may be located in a different place than in the system where XtratuM was build. In any case, it shall be the same version, than the one used to compile XtratuM.

```
$ ./xtratum-2.0.0.run
Verifying archive integrity... All good.
Uncompressing XtratuM binary distribution 2.0.0:.....

Starting installation.
Installation log in: /tmp/xtratum-installer.log

1. Select the directory where XtratuM will be installed. The installation
   directory shall not exist.

2. Select the target compiler toolchain binary directory (arch arm).

3. Confirm the installation settings.

Important: you need write permission in the path of the installation directory.

Continue with the installation [Y/n]? Y

Press [Enter] for the default value or enter a new one.
Press [TAB] to complete directory names.

1.- Installation directory [/opt]: /home/xmuser/xmenv
2.- Path to the arm toolchain [/opt/cross-compiler/bin/]: /opt/cross-compiler/bin

Confirm the Installation settings:
Selected installation path : /home/xmuser/xmenv
Selected toolchain path : /opt/cross-compiler/bin

3.- Perform the installation using the above settings [Y/n]? Y

Installation completed.
```

Listing 4.4: Output of the self-executable distribution file.

4.5 Compile the Hello World! partition

1. Change to the `INSTALL_PATH/xm-examples/hello_world` directory.

980

2. Compile partitions:

```
$ make
.....
Created by "xmuser" on "xmhost" at "Wed November 3 13:21:02 CET 2012"
XM path: " /home/xmuser/xmenv/xm"

XtratuM Core:
  Version: "2.0.0"
  Arch:   "arm"
  File:   "/home/mmunoz/Escritorio/testFolder/xm_v2.0.0-release/src/
a/core/xm_core.xef"
  Sha1:   "ce71353c6f5b85c1026cb85c1eb98dc2c5b16381"
  Changed: ""

XtratuM Library:
  Version: "2.0.0"
  File:   "/home/mmunoz/Escritorio/testFolder/xm_v2.0.0-release/src/
a/user/libxm/libxm.a"
  Sha1:   "060c383a46c00cf08ec3da8c30f2153851324a8f"
  Changed: ""

XtratuM Tools:
  File:   "/home/mmunoz/Escritorio/testFolder/xm_v2.0.0-release/src/
a/user/bin/xmcparser"
  Sha1:   "d778b2a8268b5a366f2c4fb95e20ea7545bd9bd0"
```

Note that the compilation is quite verbose: the compilation commands, messages, detailed information about the tools libraries used, etc. are printed.

The result from the compilation is a file called “resident.sw”.

Chapter 5

Partition Programming

This chapter explains how to build a XtratuM partition: partition developer tutorial. A quick introduction about how to build, load and run an example hello world partition is shown in [XM-ARM: Software Starter Guide\[1\]](#)

985

5.1 Partition definition

A partition is an execution environment managed by the hypervisor which uses the virtualised services.

Each partition consists of one or more concurrent processes (implemented by the operating system of each partition), sharing access to processor resources based upon the requirements of the application. The partition code can be:

990

- An application compiled to be executed on a bare-machine (bare-application).
- A real-time operating system and its applications.
- A general purpose operating system and its applications.

Partitions need to be *virtualised* to be executed on top of XtratuM. For instance, the partitions cannot manage directly the hardware timers. Instead XtratuM provides a set of hypercalls¹ to ask the hypervisor to enable/disable the timer and get/set their count.

995

Depending on the type of execution environment, the virtualisation implies:

Bare application An application that has been designed to run directly on the hardware could execute most of its code. Some of the hardware resources are not available directly and must be accessed using the virtualised services provided by XtratuM.

1000

Operating system application When the application runs on top of a (real-time) operating system, it uses the services provided by the operating system and does not need to be virtualised. But the operating system has to deal with the virtualisation. The operating system has to be virtualised (ported on top of XtratuM).

¹para-virtualised operations provided by the hypervisor

5.2 XAL development environment

1005 XAL is a minimal developing environment to create bare “C” applications. It is provided jointly with the XtratuM core. Currently it is only the minimal libraries and scripts to compile and link a “C” application. More features will be added in the future (mathematic lib, etc.).

In the previous versions of XtratuM, XAL was included as part of the examples of XtratuM. It has been moved outside the tree of XtratuM to create an independent developer environment.

1010 When XtratuM is installed, the XAL environment is also installed. It is included in the target directory of the installation path.

```
target_directory
|-- xal                # XAL components
|-- xal-examples      # examples of XAL use
'-- xm
```

Listing 5.1: Installation tree.

The XAL subtree contains the following elements:

```
xal
|-- arm                # Architecture Specific files
|   |-- boot.S
|   |-- defconfig
|   |-- gic.c
|   |-- init.c
|   |-- irqs.c
|   |-- Kconfig
|   '--- loader.lds
|-- bin                # Utilities
|   |-- setenv.sh
|   |-- xalinstall
|   |-- xpath
|   '--- xpathstart
|-- common             # Compilation rules and non Arch dependant files
|   |-- config.mk.dist
|   |-- hdr.c
|   |-- irqs.c
|   |-- Kconfig
|   |-- rules.mk
|   '--- std_c.c
|-- examples           # Some XAL Examples
|   |-- example.XXX
|   |   |-- Makefile
|   |   |-- partitionX.c
|   |   '--- xm_cf.arm.xml
|   '--- Makefile
|-- include            # Headers
|   |-- arm
|   |   |-- gic.h
|   |   |-- irqs.h
|   |   '--- xal.h
|   |-- assert.h
|   |-- config
|   |-- config.h
|   |-- ctype.h
|   |-- irqs.h
|   |-- limits.h
|   '--- stdarg.h
```



```

|      |-- stddef.h
|      |-- stdio.h
|      |-- stdlib.h
|      |-- string.h
|      '-- xal.h
|-- lib          # Libraries
'-- Makefile

```

Listing 5.2: XAL subtree.

A XAL partition can:

- Be specified as "system" or "user".
- Use all the XtratuM hypercalls according to the type of partition.
- Use the standard input/output "C" functions: `printf`, `sprintf`, etc. The available functions are defined in the `include/stdio.h`.

1015

5.2.1 XAL Partition Initialization

During the initialization of a Xal partition the following actions are performed:

- Initialization of the libxm.
- Install interrupts and exceptions handlers by default.
- Start virtual CPUs diferent than 0.
- Automatically enable FPU if the partition fp flag is set.

1020

5.2.2 Facilities provided by XAL

Table 5.1 shows a list of facilities provided by Xal.

	Sevice Name	Description
Interrupts:	HwCli()	Disable Interrupts
	HwSti()	Enable Interrupts
	InstallDataAbortHandler(exceptionHandler_t handler)	Install Handler
	InstallPrefetchAbortHandler(exceptionHandler_t handler)	Install Handler
	InstallUndefInstrHandler(exceptionHandler_t handler)	Install Handler
	InstallIrqHandler(xm_s32_t irqNr, exceptionHandler_t handler)	Install Handler
	InstallSwiHandler(xm_s32_t swiNr, swiHandler_t handler)	Install Handler
	InstallFPFault(exceptionHandler_t handler)	Install Handler
FPU:	FPU_ENABLE()	Enable FPU
	FPU_DISABLE()	Disable FPU

Table 5.1: Facilities provided by XAL

5.2.3 XAL Partition Example

An example of a Xal partition is:

```

#include <xm.h>
#include <stdio.h>

#define LIMIT 100

```

```

void SpentTime(int n) {
    int i,j;
    int x,y = 1;
    for (i= 0; i <=n; i++) {
        for (j= 0; j <=n; j++) {
            x = x + x - y;
        }
    }
}

void PartitionMain(void) {
    long counter=0;

    printf("[P%d] XAL Partition \n",XM_PARTITION_SELF);
    counter=1;
    while(1) {
        counter++;
        SpentTime(2000);
        printf("[P%d] Counter %d \n",XM_PARTITION_SELF, counter);
    }
    XM_halt_partition(XM_PARTITION_SELF);
}

```

Listing 5.3: XAL partition example.

1025 In the xal-examples subtree, it is possible to find several examples of XAL partitions and how these examples can be compiled. A sample Makefile containing all the compilation steps might be:

```

include $(XAL_PATH)/common/config.mk
include $(XTRATUM_PATH)/xmconfig

# XMLCF: path to the XML configuration file
XMLCF=xm_cf.$(ARCH).xml

# PARTITIONS: partition files (xef format) composing the example
SRCS := $(sort $(wildcard *.c))
OBJS := $(patsubst %.c,%.o, $(SRCS)) $(patsubst %.S,%.o, $(ASRCS))

PARTITIONS=partition0.xef partition1.xef

all: kernel.out
include $(XAL_PATH)/$(ARCH)/rules.mk

# duplicated required to run hello_world
# but should be in ../common/rules.mk
xpathstart = $(shell $(XPATHSTART) $(1) $(2))

partition0: $(OBJS)
    $(TARGET_LD) -o $$@ $^ $(TARGET_LDFLAGS) -Ttext=0x40000 -Tdata=0x8024000

partition1: $(OBJS)
    $(TARGET_LD) -o $$@ $^ $(TARGET_LDFLAGS) -Ttext=0x44000 -Tdata=0x8028000

KERNEL_ARGS=$(XMCORE_BIN)@0x0 \
    xm_cf.bin.xmc \
    partition0.bin@0x40000 \
    partition1.bin@0x44000

kernel.out: $(PARTITIONS) xm_cf.bin.xmc

```

```
$(ELFHDR) $(KERNEL_ARGS) $@
```

Listing 5.4: Makefile.

5.3 Partition reset

A partition reset is an unconditional jump to the partition entry point. There are two modes to reset a partition (see section 2.1):

- `XM_WARM_RESET`
- `XM_COLD_RESET`

1030

On a `warm reset`, the field `resetCounter` of the PCT is incremented.

On a `cold reset`: the PCT table is rebuild; `resetCounter` field is set to zero, the communication ports are closed and the timers are disarmed.

The value of `resetStatus` is:

- `0x0` when the partition is reset as result of the first time the system starts or a system reset is performed by the `XM_reset_system()` hypercall or by HM event that caused the system reset.
- `0x0` when the partition is cold reset as result of a HM event.
- `0x1` when the partition is warm reset as result of a HM event.
- Any value as provided by the `XM_partition_reset()` hypercall when the partition is reset as result of this service.

1035

1040

5.4 System reset

There are two different system reset sequences:

Warm reset: XtratuM jumps to its entry point. This is basically a software reset.

Cold reset: Execution jumps to the RSW entry point and loads as if a hardware reset had occurred.

5.5 Scheduling

5.5.1 Slot identification

A partition can get information about which is the current slot being executed querying its PCT. This information can be used to synchronize the operation of the partition with the scheduling plan.

1045

The information provided is:

Slot duration: The duration of the current slot. The value of the attribute “duration” for the current slot.

Slot number: The slot position in the system plan, starting in zero.

Id value: Each slot in the configuration file has a required attribute, named “id”, which can be used to label each slot with a user defined number.

1050

The id field is not interpreted by XtratuM and can be used to mark, for example, the slots at the starts of each period.

5.5.2 Managing scheduling plans

A system partition can request a plan switch at any time using the hypercall `XM_set_plan()`. The system will change to the new plan at the end of the current MAF. If `XM_set_plan()` is called several times before the end of the current plan, then the plan specified in the last call will take effect.

The hypercall `XM_get_plan_status()` returns information about the plan. The `xmPlanStatus_t` contains the following fields:

```
typedef struct {
    xmTime_t switchTime;
    xm_s32_t next;
    xm_s32_t current;
    xm_s32_t prev;
} xmPlanStatus_t;
```

Listing 5.5: core/include/objects/status.h

switchTime: The absolute time of the last plan switch request. After a reset (both warm and cold) its value is set to zero.

current: Identifier of the current plan.

next: The plan identifier that will be active on the next major frame. If no plan switch is going to occur, then the value of `next` is equal to the value of `current`.

prev: The identifier of the plan executed before the current one. After a reset (both warm and cold) the value is set to `(-1)`.

5.6 Console output

XtratuM offers a basic service to print a string on the console. This service is provided through a hypercall.

```
XM_write_console("Partition 1: Start execution\n", 29);
```

Listing 5.6: Simple hypercall invocation.

Additionally to this low level hypercall, some functions have been created to facilitate the use of the console by the partitions. These functions are coded in `examples/common/std.c.c`. Some of these functions are: `strlen()`, `print_str()`, `xprintf()` which are similar to the functions provided by `stdio` standard “C” library.

The use of `xprintf()` is illustrated in the next example:

```
#include <xm.h>
#include "std_c.h"    // header of the std.c.h

void PartitionMain () {    // partition entry point
    int counter=0;

    while(1) {
        counter++;
        if (!(counter%1000))
            xprintf("%d\n", counter);
    }
}
```

Listing 5.7: Ported dummy code 1

`xprintf()` performs some format management in the function parameters and invokes the hypercall which stores it in a kernel buffer. This buffer can be sent to the serial output or other device.

5.7 Inter-partition communication

Partitions can send/receive messages to/from other partitions. The basic mechanisms provided are sampling and queuing ports. The use of sampling ports is detailed in this section. 1080

Ports need to be defined in the system configuration file XM_CF. Source and destination ports are connected through channels. Assuming that ports and channel linking the ports are defined in the configuration file, the next partition code shows how to use it.

XM_create_sampling_port() and XM_create_queuing_port() hypercalls return *object descriptors* . An object descriptor is an integer value where the 16 least significant bits contains a unique identifier of the port and the 16 most significant bits are reserved for internal use. 1085

In this example partition_1 writes values in the port1 whereas partition_2 reads them.

```
#include <xm.h>
#include "std_c.h"

#define PORT_NAME "port1"
#define PORT_SIZE 48

void PartitionMain () { // partition entry point
    int counter=0;
    int portDesc;

    portDesc=XM_create_sampling_port(PORT_NAME,
                                     PORT_SIZE,
                                     XM_SOURCE_PORT);

    if ( portDesc < 0 ) {
        xprintf("[%s] cannot be created", PORT_NAME);
        return;
    }

    while(1) {
        counter++;
        if ( !(counter%1000) ){
            XM_write_sampling_message(portDesc,
                                     counter, sizeof(counter));
        }
    }
}
```

Listing 5.8: Partition_1

```
#include <xm.h>
#include "std_c.h"

#define PORT_NAME "port2"
#define PORT_SIZE 48

void PartitionMain () { // partition entry point
    int value;
    int previous = 0;
    int portDesc;
    xm_u32_t flags;

    portDesc=XM_create_sampling_port(PORT_NAME,
                                     PORT_SIZE,
                                     XM_DESTINATION_PORT);

    if ( portDesc < 0 ) {
        xprintf("[%s] cannot be created", PORT_NAME);
        return;
    }

    while(1) {
        XM_read_sampling_message(portDesc,
                                &value,
                                sizeof(value),
                                &flags);

        if (!(value == previous)){
            xprintf("%d\n", value);
            previous = value;
        }
    }
}
```

Listing 5.9: Partition_2

An interesting exercise is to determine which values will be printed.

5.7.1 Message notification

When a message is sent into a queuing port, or written into a sampling port, XtratuM triggers the extended interrupt XM_VT_EXT_OBJDESC. By default, this interrupt is masked when the partition boots. 1090

5.8 Peripheral programming

ARM CORTEX-A9 allows two different methods for performing hardware input and output operations to the peripherals:

- Mapping the peripheral as a memory area and accessing it directly as a memory-mapped I/O.
- Creating one or more IOPort(s) that allows the partition access the device.

5.8.1 Direct Memory-mapped I/O

The memory area corresponding to the device memory map is assigned to the partition in the XM_CF. Access to devices is performed directly on the bare metal without paravirtualization. The minimum memory area size assigned depends on the Page Size (5.15). The XM_CF has to specify the interrupt lines that will be used by each partition. If the device has TZ restrictions, User/Partition is in charge to configure properly the access from non-secure world.

This configuration allows the access to the device without overload.

As an additional feature XtratuM provides a easy configuration way for APB devices. The table 5.3 shows the tags used to declare devices and the physical and virtual memory base addresses where the device is placed.

Device	Base Address	Device	Base Address
UART 0	0xE0000000	UART 1	0xE0001000
USB 0	0xE0002000	USB 1	0xE0003000
I2C 0	0xE0004000	I2C 1	0xE0005000
SPI 0	0xE0006000	SPI 1	0xE0007000
CAN 0	0xE0008000	CAN 1	0xE0009000
GPIO	0xE000A000	GEM 0	0xE000B000
GEM 1	0xE000C000	QSPI	0xE000D000
SMC	0xE000E000	TRIPLE TIMER	0xF8002000

Table 5.3: List of ZYNQ APB Device Names.

XtratuM maps in user space the address range corresponding to each device when this is configured in XM_CF. For mor detailed informations reference section 5.15. Extra information about device configuration is find in section 8.2.6.

5.8.2 Device is accessed by IOPorts

Access to devices is performed using XtratuM IOPorts. During the service access to the device is performed with XtratuM permissions therefore TZ has no influence if this method is used. The service is offered through two hypercalls to access I/O registers: XM_arm_inport() and XM_outport(). In order to be able to access (read from or write to) hardware I/O port the corresponding ports have to be allocated to the partition in the XM_CF configuration file. XtratuM is responsible for mapping the areas of memory where the device is placed. This memory map is not accessible for partitions. Two partitions can not use neither the same device nor a interrupt line.

There are two methods to allocate ports to a partition in the configuration file:

Range of ports: A range of I/O ports, with no restriction, allocated to the partition. The Range element is used.

Restricted port: A single I/O port with restrictions on the values that the partition is allowed to write in. The Restricted element is used in the configuration file. There are two kind of restrictions that can be specified:

Bitmask: Only those bits that are set, can be modified by the partition. In the case of a read operation only those bits set in the mask will be returned to the partition; the rest of the bits will be resetted. Attribute mask.

The attribute (mask is optional. A restricted port declaration with no attribute, is equivalent to declare a range of ports of size one. In the case that both, the bitmap and the range of values, are specified then the bitmap is applied first and then the range is checked.

The implementation of the bit mask is done as follows:

```
oldValue=LoadIoReg(port);
if((port>=CA9_SLCR_BASE_ADDRESS)&&(port<=(CA9_SLCR_BASE_ADDRESS+0
x00000B74)))
{
    CA9_TZ_UNLOCK_SLCR(); }
StoreIoReg(port, ((oldValue&~(mask))|(value&mask)));
if((port>=CA9_SLCR_BASE_ADDRESS)&&(port<=(CA9_SLCR_BASE_ADDRESS+0
x00000B74)))
{
    CA9_TZ_LOCK_SLCR(); }
```

Listing 5.10: core/kernel/arm/hypcalls.c

First, the port is read, to get the value of the bits not allocated to the partitions, then the bits that have to be modified are changed, and finally the value is written back.

The read operation shall not cause side effects on the associated peripheral. For example, some devices may interpret as interrupt acknowledge to read from a control port. Another source of errors may happen when the restricted is implemented as an open collector output. In this case, if the pin is connected to an external circuit which forces a low voltage, then the value read from the io port is not the same that the previous value written.

The following example declares a range of ports and two restricted ones.

```
<Partition ..... >
  <HwResources>
    <IoPorts>
      <Restricted address="0xf8f03000" mask="0xff" />
    </IoPorts>
  </HwResources>
</Partition>
```

If the bitmask restriction is used, then the bits of the port that are not set in the mask can be allocated to other partitions. This way, it is possible to perform a fine grain (bit level) port allocation to partitions. That is a single ports can be safely shared among several partitions.

5.8.3 FPGA Partition Access

TrustZone allows to enable or disable the access to FPGA addresses from “non-secure World” (partitions). Access to FPGA addresses from Secure World (hypervisor) is always allowed.

XtratuM provides a Hypervisor Feature in the configuration file (XM.CF) to allow the “non-secure World” access (i.e. partitions) to FPGA memory regions. Details about how to configure Hypervisor Feature are shown in section 8.2.3.

This option have been included in the XM.CF to offer flexibility in the building of the system.

When this hypervisor feature is enabled, both M_AXI_GP0 and M_AXI_GP1 slave ports are accessible from “non-secure World”.

At any case XtratuM provides two hypercalls to access I/O registers: XM_arm_inport() and XM__outport(). In order to be able to access (read from or write to) hardware I/O port the corresponding ports have to be allocated to the partition in the XM.CF configuration file.

NOTE:

To allow Partition access to Programming Logic (PL) is mandatory to configure some general purpose AXI interface clocks (ACLK signal) on Zynq processing system7 block. This is needed in order to allow the Hypervisor to configuration the TrustZone registers that manages the PL access.

1160 The PL configuration must provide clock to “S_AXI_GPO_ACLK” and “S_AXI_GPI_ACLK” nets. This clock must be provided even if those nets were unused. A solution for this is to loop the FCLK0 clock back to those clock nets.

Additionally the bitstream must be loaded into the Zynq FPGA before to start the XtratuM hypervisor.

Use case example:

This example describes how to configure and use this feature.

1165 The XM_CF must contains the “XM_HYP_FEAT_FPGA_PART_ACCESS” feature enabled as shown in the following xml extract:

```
<XMHypervisor console="Uart" features="XM_HYP_FEAT_FPGA_PART_ACCESS" >
  <PhysicalMemoryArea size="512KB" />
</XMHypervisor>
```

Additionally the partition must configure the memory areas needed to access the FPGA areas as needed.

As example if a partition needs to access the FPGA in address range “[0x41200000-0x41200FFF]” the partition configuration area looks like:

```
<Partition id="0" name="Partition0" flags="boot" console="Uart" >
  <PhysicalMemoryAreas>
    <Area start="0x10000000" size="256KB" />
    <Area start="0x41200000" size="4KB" /> <!-- FPGA IO registers-->
  </PhysicalMemoryAreas>
</Partition>
```

Four different scenarios could be observed from the combination of these configurations.

1170 Scenario 1:

Used configuration:

Hypervisor defines the “FPGA_PART_ACCESS” feature.

Partition uses direct access to FPGA IO registers. e.g. `< Areastart = "0x41200000" size = "4KB" / >`

FPGA does not configure the ACLK interfaces such as it is expected.

1175

Result: As result of this scenario the system crashes at boot time. System crash means that XtratuM does not run, the system/board become frozen, the connection with XMD is missed, etc). In order to fix this issue the scenario must remove the use of the FPGA_PART_ACCESS hypervisor feature in the XM_CF and the partition must use XtratuM I/O ports services and remove the memory area.

1180 Scenario 2:

Used configuration:

Hypervisor does not define the “FPGA_PART_ACCESS” feature.

Partition uses direct access to FPGA IO registers. e.g. `< Areastart = "0x41200000" size = "4KB" / >`

1185 FPGA does not configure the ACLK interfaces such as it is expected or FPGA configures the ACLK interfaces such as it is expected.

Result: A DATA_ABORT exception is raised at partition level when the partition try to access to FPGA address. In order to fix this issue:

1190 If FPGA does not configure the ACLK interfaces: The partition must use XM IO port services (remove direct access from the XML and use inport/outport).

If FPGA configures the ACLK interfaces such as it is expected: add the FPGA_PART_ACCESS hypervisor feature in the XM_CF or the partition could use XtratuM I/O ports services and remove the memory area.

Scenario 3:**Used configuration:**

1195

Hypervisor defines the “FPGA.PART.ACCESS” feature.
 Partition uses direct access to FPGA IO registers. e.g. `< Areastart = "0x41200000" size = "4KB" / >`
 FPGA configures properly the ACLK interfaces.

Result: The access to FPGA succeed.

1200

Scenario 4:**Used configuration:**

Hypervisor defines the “FPGA.PART.ACCESS” feature.
 Partition uses both access types: XM IO ports services and direct access to FPGA IO registers.
 FPGA configures properly the ACLK interfaces.

1205

Result: The access to FPGA succeed.

5.8.4 PMU Partition Access

The Performance Monitor Unit (PMU) can be accessed from “non-secure World” (partitions). The partition is in charge of enabling and configuring the PMU. XtratuM does not reset and maintain the state of the PMU at hypervisor initialization time nor between the execution of partitions, when a context switch takes place.

1210

The partition shall configure the Performance monitor registers (c9 registers) in order to enable/disable the access to PMU and, configure and control the counters for each processor.

Note that the PMCR register, used to enable/disable the PMU, is a non-banked register, therefore this register is shared among the partitions of the system.

5.9 Exceptions

For each partition, XtratuM defines the exception vector by default at the beginning of its code memory map. An extend explanation about how exceptions are served is done in section 2.11.

1215

5.9.1 Interrupts/Fast interrupts

In order to properly manage a peripheral, a partition can request to manage directly a hardware interrupt line. To do so, the interrupt line shall be allocated to the partition in the configuration file.

On ARM CORTEX-A9 architecture, XtratuM supports up to 96 hardware interrupts. They are extended with 32 additional fake interrupts that are generated by the hypervisor itself, and that provide a mean to notify the partition about some events related with the virtualisation or other services provided by the hypervisor. The list of extended interrupts, which also contains its description, can be found in table 5.4. It is not necessary to allocate extended interrupts to partitions, since each partition can receive them all.

1220

Hardware interrupts, as well as extended interrupts, can be individually enabled/disabled by means of the `XM_set_irqmask()` and `XM_clear_irqmask()` hypercalls. Partitions can also enable/disable the interrupts as a whole by means of the appropriate flags in the CPSR register. Changing the CPSR flags associated to interrupts, also affect the propagation of the extended interrupts to the partition.

1225

XtratuM allows the partitions to clear the pending interrupts by means of the `XM_clear_irqpend()` hypercall. This applies to both the hardware and the extended interrupts. Additionally, a partition can request XtratuM to simulate the arrival of **hardware** interrupts for itself using the `XM_set_irqpend()` service.

1230

Finally, when a extended IRQ exception is taken (emulated), XtratuM will provide the number of interrupt triggering the exception in the `irqIndex` field of the *Partition Control Table* (see 6.5.2 while the real `Irq Index`

IRQ offset	Mnemonic (C macro)	Description
0	<code>XM_VT_EXT_HW_TIMER</code>	The timer associated with the hardware clock (<code>XM_HW_CLOCK</code>) has expired
1	<code>XM_VT_EXT_EXEC_TIMER</code>	The timer associated with the execution clock (<code>XM_EXEC_CLOCK</code>) has expired
4	<code>XM_VT_EXT_SAMPLING_PORT</code>	A message has been received at an input <i>sampling</i> port
5	<code>XM_VT_EXT_QUEUING_PORT</code>	A message has been received at an input <i>queuing</i> port
8	<code>XM_VT_EXT_CYCLIC_SLOT_START</code>	The start of a new slot assigned to the partition has been reached.

Table 5.4: Extended interrupts

Register contains value 0).

5.9.2 Parity errors

A parity error is an error that results from irregular changes to data, between the time it is written to memory and the time that it is read back. Some types of parity errors do not require extensive correction, but others can damage the integrity of the system. Serious parity errors, if undetected, may cause the corruption of stored data or may have unpredictable results or a system crash.

The parity error detection is a technique for detecting and correcting parity errors. Such technique uses an extra bit, called a parity bit, which is set so that all bytes have either an odd number (odd parity) or an even number (even parity) of set bits. The parity bit calculated from the stored data is then compared to the final parity bit. If these two values differ, this indicates a data error, and at least one bit must have been changed due to data corruption.

The ARM CORTEX-A9 architecture provides support for parity errors detection using the even parity scheme. XtratuM provides the mechanisms to catch and recover from the parity errors exceptions.

The set of sources of parity errors and the recovery actions to carry out when a parity error is detected, is listed in the next table:

Source of parity error	Action ¹
L1 data cache	No action.
L1 instruction cache	Invalidate L1 instruction cache line.
L2 cache	No action / Invalidate L2 cache. ²
Translation Lookaside Buffer (TLB)	Invalidate TLB
Snoop Control Unit (SCU)	Invalidate SCU
Global History Buffer (GHB)	No action. Automatically detected and corrected.
Branch Target Address Cache (BTAC)	No action. Automatically detected and corrected.

Table 5.5: Parity errors.

Once a parity error is detected, XtratuM applies the corresponding recovery action depending on the type of parity error received and registers the error in the status of the system, by incrementing the counter associated to the parity error source (see 5.14). If the error is corrected the system continues running. Otherwise, if the error cannot be corrected, a data error hm event is raised.

¹Recovery action "No action" means that the error cannot be corrected and a data error hm event will be raised.

²Only invalidate when the L2 cache policy is write-through/no write-allocate to avoid data loss. Otherwise, the action to be applied is no action.

5.10 Clock and timer services

XtratuM provides the `XM_get_time()` hypercall to read the time from a clock, and the `XM_set_timer()` hypercall to arm a timer.

There are two clocks available:

```
#define XM_HW_CLOCK (0x0)
#define XM_EXEC_CLOCK (0x1)
```

1255


Listing 5.11: core/include/hypercalls.h

XtratuM provides one timer for each clock. The timers can be programmed in one-shot or in periodic mode. Upon expiration, the extended interrupts `XM_VT_EXT_HW_TIMER` and `XM_VT_EXT_EXEC_TIMER` are triggered.

5.10.1 Execution time clock

The clock `XM_EXEC_CLOCK` only advances while the partition is being executed or while XtratuM is executing a hypercall requested by the partition. The execution time clock computes the total time used by the target partition.

This clock relies on the `XM_HW_CLOCK`, and so, its resolution is also $1\mu\text{sec}$. Its precision is not as accurate as that of the `XM_HW_CLOCK` due to the errors introduced by the partition switch. 1260

The execution time clock does not advance when the partition gets idle or suspended. Therefore, the `XM_EXEC_CLOCK` clock should not be used to arm a timer to wake up a partition from an idle state. 

The code below computes the temporal cost of a block of code.

```
#include <xm.h>
#include "std_c.h"

void PartitionMain() {
    xmTime_t t1, t2;

    XM_get_time(XM_EXEC_CLOCK, &t1);
    // code to be measured
    XM_get_time(XM_EXEC_CLOCK, &t2);
    xprintf("Initial time: %lld, final time: %lld", t1, t2);
    xprintf("Difference: %lld\n", t2-t1);
    XM_halt_partition(XM_PARTITION_SELF);
}
```

5.11 Processor management

Most of the registers are banked between the User and Hypervisor “modes”. This avoid to use XtratuM services to access them. 1265

5.11.1 Current Program Status Register

The contents of both the CPSR and the SPSR is accesible directly by the user. Some bits in these registers are read-only for the user as the Fast Interrupt Disable Flag.

5.11.2 Processor Modes

Most of the Processor Modes are available for the user at partition level.

Name	Abrev	Encoding	User available
USER	usr	0x10	Yes
FIQ	fiq	0x11	Yes
IRQ	irq	0x12	Yes
SUPERVISOR	svc	0x13	Yes
MONITOR	mon	0x16	NO
ABORT	abt	0x17	Yes
UNDEFINED	und	0x1B	Yes
SYSTEM	sys	0x1F	Yes

5.12 Synchronization

1270 There are two different ways to launch a new MAF in XtratuM:

1. in the context switch, XtratuM, the scheduler automatically launches a new MAF when the last MAF is nnished.
2. XtratuM by external synchronization (if enabled in the menucong).

5.13 Tracing

5.13.1 Trace messages

1275 The hypercall `XM_trace_event()` stores a trace message in the partition's associated buffer. A trace message is a `xmTraceStatus_t` structure which contains a *timestamp* and an associated user defined data:

```

1280 struct xmTraceEvent {
        xm_u32_t timestamp; // LSB of time
        xm_u8_t payload[XM_TRACE_PAYLOAD_LENGTH];
    } __PACKED;

    typedef struct xmTraceEvent xmTraceEvent_t;

```

Listing 5.12: core/include/objects/trace.h

1285 **timestamp:** A time stamp of when the trace was generated.

payload: Used defined data.

5.13.2 Reading traces

Only one system partition can read from a trace stream. A standard partition **can not read its own trace messages**, it is only allowed to store traces on it.

Configuration

1290 XtratuM statically allocates blocks of memory to store all traces. The amount of memory reserved to store traces is a configuration parameter of the sources (see section 8.1). Particularly, the number of traces stored per partition is configured in [Objects - Size of the XM's internal buffer to store trace logs]

5.14 System and partition status

The hypercalls `XM_get_partition_status()` and `XM_get_system_status()` return information about a given partition and the system respectively.

The data structure returned are:

```

typedef struct {
    /* Current state of the partition: ready, suspended ... */
    xm_u32_t state[CONFIG_NO_VCPUS];
#define XM_STATUS_IDLE 0x0
#define XM_STATUS_READY 0x1
#define XM_STATUS_SUSPENDED 0x2
#define XM_STATUS_HALTED 0x3
    /* Number of virtual interrupts received. */
    xm_u64_t noVirqs; /* [[OPTIONAL]] */
    /* Reset information */
    xm_u32_t resetCounter;
    xm_u32_t resetStatus;
    xmTime_t execClock[CONFIG_NO_VCPUS];
    /* Total number of partition messages: */
    xm_u64_t noSamplingPortMsgsRead; /* [[OPTIONAL]] */
    xm_u64_t noSamplingPortMsgsWritten; /* [[OPTIONAL]] */
    xm_u64_t noQueuingPortMsgsSent; /* [[OPTIONAL]] */
    xm_u64_t noQueuingPortMsgsReceived; /* [[OPTIONAL]] */
    xmTime_t irqLastOccurrence[CONFIG_NO_HWIRQS];
} xmPartitionStatus_t;

```

Listing 5.13: core/include/objects/status.h

```

typedef struct {
    xm_u32_t resetStatus;
    xm_u32_t resetCounter;
    /* Number of HM events emitted. */
    xm_u64_t noHmEvents; /* [[OPTIONAL]] */
    /* Number of HW interrupts received. */
    xm_u64_t noIrqs; /* [[OPTIONAL]] */
    /* Current major cycle iteration. */
    xm_u64_t currentMaf; /* [[OPTIONAL]] */
    /* Total number of system messages: */
    xm_u64_t noSamplingPortMsgsRead; /* [[OPTIONAL]] */
    xm_u64_t noSamplingPortMsgsWritten; /* [[OPTIONAL]] */
    xm_u64_t noQueuingPortMsgsSent; /* [[OPTIONAL]] */
    xm_u64_t noQueuingPortMsgsReceived; /* [[OPTIONAL]] */
    xmTime_t irqLastOccurrence[CONFIG_NO_HWIRQS];
    /* Total number of parity errors: */
    xm_u64_t noL1ICacheParityErrors;
    xm_u64_t noL1DCacheParityErrors;
    xm_u64_t noTLBParityErrors;
    xm_u64_t noL2CacheParityErrors;
    xm_u64_t noSCUParityErrors;
} xmSystemStatus_t;

```

Listing 5.14: core/include/objects/status.h

The field `execClock` of a partition is the execution time clock of the target partition. The rest of the fields are self explained.

Those fields commented as `[[OPTIONAL]]` contain valid data only if XtratuM has been compiled with the flag “Enable system/partition status accounting” enabled.

5.15 Memory management



1345 XtratuM implements a flat memory space on the architecture. The addresses generated by the control unit are directly emitted to the memory controller without any translation. Therefore, **each partition shall be compiled and linked to work on the designated memory range**. The starting address and the size of each partition is specified in the system configuration file.

1350 MMU is used to implement memory protection. XtratuM will map the memory areas allocated to each partition using the page size that better fits the memory areas sizes. The available page sizes depends on the architecture. The listing 5.15 shows the sizes used currently.

XtratuM manages the MMU and Page Tables.

```
1355 #define PAGE_SIZE (4*1024)    /*4kB page*/
    #define LPAGE_SIZE (1*1024*1024) /*1MB section*/
```

Listing 5.15: core/include/arm/paging.h

5.15.1 Partition Memory Map

The partition memory map is the set of memory areas accessible directly by the partition without need of use an XtratuM service. The partition virtual memory map is static from the partition point of view.

1360 The partition memory map is formed by the memory areas defined in the XM_CF. Additionally XtratuM provides access to a Partition Control Table and those devices assigned to each partition (e.g.: Interrupt Controller, assigned APB Devices, ...).

The partition virtual memory map consists of:

MMap Area	Condition
Partition Memory Areas	defined in XM_CF
Partition Control Table	Always
Interrupt Controller	Always
APB Device	If defined in XM_CF

Table 5.6: Partition Memory Map.

5.15.2 Memory access restrictions

1365 Trust Zone technology provides the possibility to limit the memory map showed to the partition, independently of the memory map. As example, the memory map of the UART is not accessible from user space, even if the UART memory map is added in XM_CF, if the UART Trust Zone configuration is not set properly. The Figure 5.1 depicts this feature.

1370 Some registers in the Zynq can be accesed directly through direct memory-mapped I/O, but in some other cases it is not possible due to Trust Zone restrictions. Then, it is required to access the register memory area from the Secure world, that is, through I/O ports (see 5.8.2). The type of access available for each Zynq memory-mapped register is summarized in the table 5.7.

	Address	Description	Is accessible through I/O Ports?	Is accessible through direct Memory-mapped I/O?
I/O Peripheral:	0xE0000000	UART Controller 0	Yes	Yes ²
	0xE0001000	UART Controller 1	Yes	Yes ²
	0xE0002000	USB Controller 0	Yes	Yes ²
	0xE0003000	USB Controller 1	Yes	Yes ²
	0xE0004000	I2C Controller 0	Yes	Yes ²
	0xE0005000	I2C Controller 1	Yes	Yes ²
	0xE0006000	SPI Controller 0	Yes	Yes ²
	0xE0007000	SPI Controller 1	Yes	Yes ²
	0xE0008000	CAN Controller 0	Yes	Yes ²
	0xE0009000	CAN Controller 1	Yes	Yes ²
	0xE000A000	GPIO Controller	Yes	Yes ²
	0xE000B000	Ethernet Controller 0	Yes	Yes ²
	0xE000C000	Ethernet Controller 1	Yes	Yes ²
	0xE000D000	Quad-SPI Controller	Yes	Yes ²
	0xE000E000	Static Memory Controller (SMC)	Yes	Yes ²
	0xE0100000	SDIO Controller 0	Yes	No
	0xE0101000	SDIO Controller 1	Yes	No
SLCR:	0xF8000000	SLCR write protection lock and security	Yes	No
	0xF8000100	Clock control and status	Yes	No
	0xF8000200	Reset control and status	Yes	No
	0xF8000300	APU control	Yes	No
	0xF8000400	TrustZone control	Yes	No
	0xF8000500	CoreSight SoC debug control	Yes	No
	0xF8000600	DDR DRAM controller	Yes	No
	0xF8000700	MIO pin configuration	Yes	No
	0xF8000800	MIO parallel access	Yes	No
	0xF8000900	Miscellaneous control	Yes	No
	0xF8000A00	On-chip memory (OCM) control	Yes	No
	0xF8000B00	I/O buffers for MIO(GPIOB)/DDR(DDRIOB)	Yes	No
PS System:	0xF8001000	Triple timer counter 0 (TTC 0)	Yes	No
	0xF8002000	Triple timer counter 1 (TTC 1)	Yes	No
	0xF8003000	DMAC when secure (DMAC S)	Yes	No
	0xF8004000	DMAC when non-secure (DMAC NS)	Yes	Yes
	0xF8005000	System watchdog timer (SWDT)	Yes	No
	0xF8006000	DDR memory controller	Yes	No
	0xF8007000	Device configuration interface (DevC)	Yes	No
	0xF8008000	AXI_HP 0 high performance	Yes	No
	0xF8009000	AXI_HP 1 high performance	Yes	No
	0xF800A000	AXI_HP 2 high performance	Yes	No
	0xF800B000	AXI_HP 3 high performance	Yes	No
	0xF800C000	On-chip memory (OCM)	Yes	No
	0xF8800000	CoreSight debug control	Yes	No
CPU Private:	0xF8900000	Top-level interconnect configuration and Global Programmers View (GPV)	Yes	No
	0xF8F00000	SCU control and status	Yes	Yes ³
	0xF8F00100	Interrupt controller CPU	Yes	Yes ³
	0xF8F00200	Global timer	Yes	Yes ^{3,4}
	0xF8F00600	Private timers and private watchdog timers	Yes	Yes ^{3,4}
	0xF8F01000	Interrupt controller distributor	Yes	Yes ^{3,4}
	0xF8F02000	L2-cache controller	Yes	No

Table 5.7: Memory-mapped registers access.

²The corresponding APB device must be assigned to the partition, its memory area is automatically generated, please refer to section 5.8.1.

³XtratuM defines this memory area for each partition by default, therefore it is not necessary to be mapped by the partition.

⁴Some registers are banked between Secure and Non-Secure world.

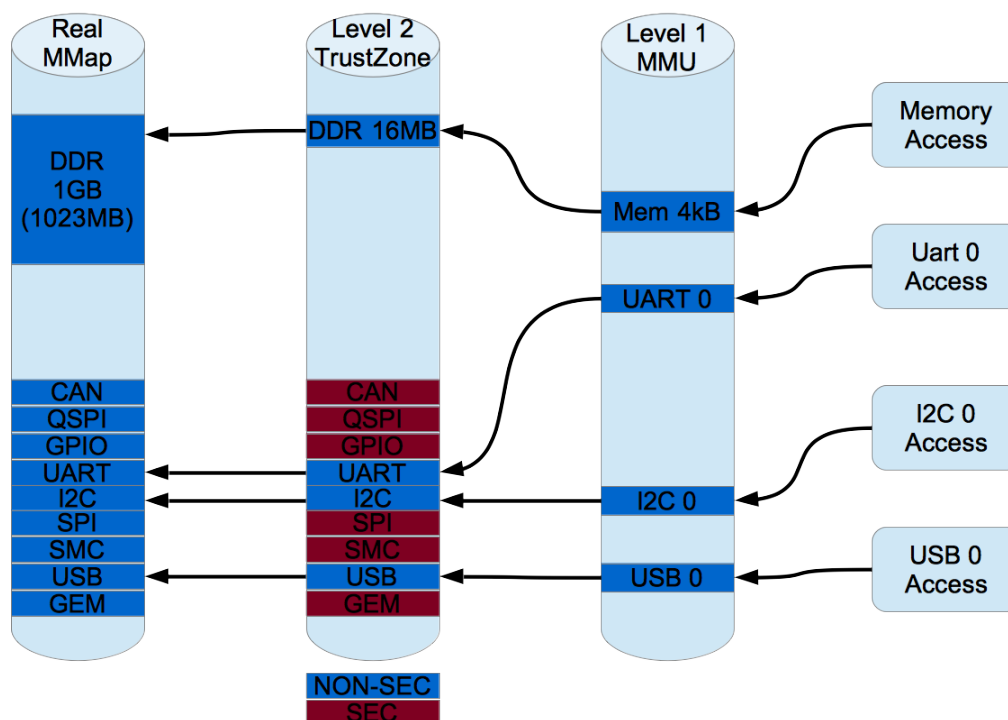


Figure 5.1: Trust Zone memory access model.

5.15.3 Configuration

A partition can statically allocate several memory areas to a partition. Memory areas have to be properly configured in the XM_CF configuration file in a coherent way with respect to the memory available and the allocation of memory areas to XtratuM, container, bootloader and other partitions.

5.16 Releasing the processor

In some situations, a partition is waiting for a new event to execute a task. If no more tasks are pending to be executed, then the partition can become idle. The idle partition becomes ready again when an interrupt is received.

The partition can inform XtratuM about its idle state (see `XM.idle.self()`). In the current implementation XtratuM does nothing while a partition is idle, that is, no other partition is executed; it opens the possibility to take advantage of the unused time for internal maintenance activities or to apply an energy saving policy.

Since XtratuM delivers an event on every new slot, the idle feature can also be used to synchronize the operation of the partition with the scheduling plan.

5.17 Partition customisation files

A partition is composed of a binary image (code and data) and, zero or more additional files (customisation files). To ease the management of these additional files, the header of the partition image (see section 6.5.1) holds the fields `noModules` and `moduleTab`, where the first is the number of additional files which have to be loaded and the second is an array of data structures which define the loading address and the sizes of these additional files. During its creation, the partition is responsible for filling these fields with the address of a pre-allocated memory area inside its memory space.

This information shall be used by the loader software, for instance the resident software or a manager system

partition, in order to know the place where to copy into RAM these additional files. If the size of any of these files is larger than the one specified on the header of the partition or the memory address is invalid, then the loading process will fail.

These additional files shall be accessible by part of the loader software. For example, they must be packed jointly with the partition binary image by using the `xmpack` tool.

1395

5.18 The object interface

XtratuM implements internally a kind of virtual file system (as the `/dev` directory). Most of the libxm hypercalls are implemented using this file system. The hypercalls to access the objects are used internally by the libxm and shall not be used by the programmer. They are listed here just for completeness:

```
extern __stdcall xm_s32_t XM_get_gid_by_name(xm_u8_t *name, xm_u32_t entity);
extern __stdcall xmId_t XM_get_vcpuid(void);

// Time management hypercalls
extern __stdcall xm_s32_t XM_get_time(xm_u32_t clock_id, xmTime_t *time);
extern __stdcall xm_s32_t XM_set_timer(xm_u32_t clock_id, xmTime_t abstime, xmTime_t
    interval);

// Partition status hypercalls
extern __stdcall xm_s32_t XM_suspend_partition(xm_u32_t partition_id);
extern __stdcall xm_s32_t XM_resume_partition(xm_u32_t partition_id);
extern __stdcall xm_s32_t XM_resume_imm_partition(xm_u32_t partition_id);
extern __stdcall xm_s32_t XM_shutdown_partition(xm_u32_t partition_id);
extern __stdcall xm_s32_t XM_reset_partition(xm_u32_t partition_id, xmAddress_t ePoint,
    xm_u32_t resetMode, xm_u32_t status);
extern __stdcall xm_s32_t XM_halt_partition(xm_u32_t partition_id);
extern __stdcall xm_s32_t XM_idle_self(void);

extern __stdcall xm_s32_t XM_reset_vcpu(xm_u32_t vcpu_id, xmAddress_t entry);
extern __stdcall xm_s32_t XM_halt_vcpu(xm_u32_t vcpu_id);

// system status hypercalls
extern __stdcall xm_s32_t XM_halt_system(void);
extern __stdcall xm_s32_t XM_reset_system(xm_u32_t resetMode);

// Object related hypercalls

extern __stdcall xm_s32_t XM_read_object(xmObjDesc_t objDesc, void *buffer, xm_u32_t size,
    xm_u32_t *flags);
extern __stdcall xm_s32_t XM_write_object(xmObjDesc_t objDesc, void *buffer, xm_u32_t size
    , xm_u32_t *flags);
extern __stdcall xm_s32_t XM_seek_object(xmObjDesc_t objDesc, xm_u32_t offset, xm_u32_t
    whence);
extern __stdcall xm_s32_t XM_ctrl_object(xmObjDesc_t objDesc, xm_u32_t cmd, void *arg);
```

1400

1405

1410

1415

1420

1425

1430

1435

Listing 5.16: `user/libxm/include/xmhypercalls.h`

The following services are implemented through the object interface:

- Communication ports.
- Console output.
- Health monitoring logs.
- Memory access.
- XtratuM and partition status.
- Trace logs.
- Serial ports.

1440

For example, the `XM_hm_status()` hypercall is implemented in the libxm as:

```
1445 xm_s32_t XM_hm_status(xmHmStatus_t *hmStatusPtr) {  
  
    if (!hmStatusPtr) {  
        return XM_INVALID_PARAM;  
    }  
1450 return XM_ctrl_object(OBJDESC_BUILD(OBJ_CLASS_HM, XM_HYPERVISOR_ID, 0),  
    XM_HM_GET_STATUS, hmStatusPtr);  
}
```

Listing 5.17: user/libxm/common/hm.c

Chapter 6

Binary Interfaces

This section covers the data types and the format of the files and data structures used by XtratuM.

1455

Only the first section, describing the data types, is needed for a partition developer. The remaining sections contain material for more advanced users. The `libxm.a` library provides a friendly interface that hides most of the low level details explained in this chapter.

6.1 Data representation

The data types used in the XtratuM interfaces are compiler and machine cross development independent. This is specially important when manipulating the configuration files. Note that the development PC and the board could differ in endianness.



1460

XtratuM conforms the following conventions:

Unsigned	Signed	Size (bytes)	Alignment (bytes)
<code>xm_u8_t</code>	<code>xm_s8_t</code>	1	1
<code>xm_u16_t</code>	<code>xm_s16_t</code>	2	4
<code>xm_u32_t</code>	<code>xm_s32_t</code>	4	4
<code>xm_u64_t</code>	<code>xm_s64_t</code>	8	8

Table 6.1: Data types.

These data types have to be stored in big-endian format, that is, the most significant byte is the rightmost byte (`0x..00`) and the least significant byte is the leftmost byte (`0x..03`).

“C” declaration which meet these definitions are presented in the list below:

1465

```
// Basic types
typedef unsigned char xm_u8_t;
typedef char xm_s8_t;
typedef unsigned short xm_u16_t;
typedef short xm_s16_t;
typedef unsigned int xm_u32_t;
typedef int xm_s32_t;
typedef unsigned long long xm_u64_t;
typedef long long xm_s64_t;
```

1470

1475

Listing 6.1: `core/include/arm/arch_types.h`

```

// Extended types
typedef long xmLong_t;
1480 typedef xm_u32_t xmWord_t;
#define XM_LOG2_WORD_SZ 5
typedef xm_s64_t xmTime_t;
#define MAX_XMTIME 0x7fffffffffffffffLL
1485 typedef xm_u32_t xmAddress_t;
typedef xmAddress_t xmIoAddress_t;
typedef xm_u32_t xmSize_t;
typedef xm_s32_t xmSSize_t;
typedef xm_u32_t xmId_t;

```

Listing 6.2: core/include/arm/arch_types.h

1490 For future compatibility, most data structures contain version information. It is a `xm_u32_t` data type with 3 fields: version, subversion and revision. The macros listed next can be used to manipulate those fields:

```

1495 #define XM_SET_VERSION(_ver, _subver, _rev) ((((_ver)&0xFF)<<16)|(((_subver)&0xFF)
    <<8)|(((_rev)&0xFF))
#define XM_GET_VERSION(_v) (((_v)>>16)&0xFF)
#define XM_GET_SUBVERSION(_v) (((_v)>>8)&0xFF)
#define XM_GET_REVISION(_v) ((_v)&0xFF)

```

Listing 6.3: core/include/xmef.h

6.2 Hypercall mechanism

1500 Hypercalls are implemented by using the SMC exception. When a partition requires to invoke an hypercall, the `smc` assembly instruction is invoked using as parameter the hypercall number being invoked. Additionally, since this exception is also used by the partition to implement system calls, a signature (0x24786879) is added just after the invocation. XtratuM starts hypercall dispatching after acknowledging such signature. Up to five parameters can be passed through the processor registers.

6.3 Assembly programming

This section describes the assembly programming convention used to implement the XtratuM hypercall mechanism.

1505 The register assignment convention for calling an hypercall is:

%r0 - %r5 Hold the parameters for the hypercall.

The return value is stored in register **%r0**.

For example, the following assembly code calls the `XM_get_time(xm_u32_t clockId, xmTime_t *time)`:

```

smc #0xa /* __GET_TIME_NR */
.long 0x24786879

```

In ARM, the `get_time_nr` constant has the value “0xa”; “%r0” holds the clock id; and “%r1” is a pointer which points to a `xmTime_t` variable. The return value of the hypercall is stored in “%r0”.

1510 Below is the list of normal hypercall number constants (listing 6.3):

```

#define __HALT_PARTITION_NR 0
#define __SUSPEND_PARTITION_NR 1
#define __RESUME_PARTITION_NR 2

```

```

#define __RESET_PARTITION_NR 3
#define __SHUTDOWN_PARTITION_NR 4
#define __HALT_SYSTEM_NR 5
#define __RESET_SYSTEM_NR 6
#define __IDLE_SELF_NR 7

#define __GET_TIME_NR 8
#define __SET_TIMER_NR 9
#define __READ_OBJECT_NR 10
#define __WRITE_OBJECT_NR 11
#define __SEEK_OBJECT_NR 12
#define __CTRL_OBJECT_NR 13

#define __CLEAR_IRQ_MASK_NR 14
#define __SET_IRQ_MASK_NR 15
#define __FORCE_IRQS_NR 16
#define __CLEAR_IRQS_NR 17
#define __ROUTE_IRQ_NR 18

#define __SET_CACHE_STATE_NR 19

#define __SWITCH_SCHED_PLAN_NR 20
#define __GET_GID_BY_NAME_NR 21

#define arm_inport_nr 23
#define arm_outport_nr 24
// #define reserved_0 29
// #define reserved_1 30

#define __RESET_VCPU_NR 31
#define __HALT_VCPU_NR 32
#define __GET_VCPUID_NR 33
#define __RAISE_IPVI_NR 34

#define arm_hm_raise_event_nr 35
#define __RESUME_IMM_PARTITION_NR 36
#define __SWITCH_IMM_SCHED_PLAN_NR 37

```

Listing 6.4: core/include/arm/hypercalls.h

The file “core/include/arm/hypercalls.h” has additional services for the ARM architecture.

6.4 Executable formats overview

XtratuM core does not have the capability to “load” partitions. It is assumed that when XtratuM starts its execution, all the partition code and data required to execute each partition is already in main memory. Therefore, XtratuM does not contain code to manage executable images. The only information required by XtratuM to execute a partition is the address of the partition image header (`xmImageHdr`).

The partition images, as well as the XtratuM image, shall be loaded by a resident software, which acts as the boot loader.

The XEF (XtratuM Executable Format) has been designed as a robust format to copy the partition code (and data) from the partition developer to the final target system.

The XtratuM image shall also be in XEF format. From the resident software point of view, XtratuM is just another image that has to be copied into the appropriate memory area.

The main features of the XEF format are:

- Simpler than the ELF. The ELF format is a rich and powerful specification, but most of its features are not required.
- Content checksum. It allows to detect transmission errors.
- Compress the content. This feature greatly reduce the space of the image; consequently the deploy time.

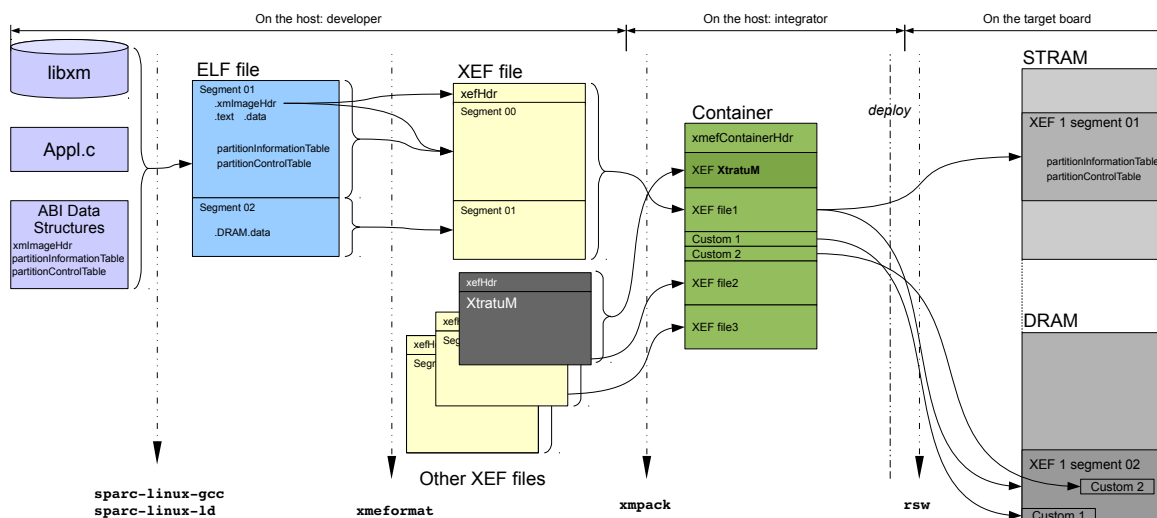


Figure 6.1: Executable formats.

- Content encryption. Not implemented.
- Partitions can be placed in several non-contiguous memory areas.

The *container* is a file which contains a set of XEF files. It is like a tar file (with important internal differences). The resident software shall be able to manage the container format to extract the partitions (XEF files) and also the XEF format to copy them to the target memory addresses.

The signature fields are constants used to identify and locate the data structures. The value that shall contain these fields on each data structure is defined right above the corresponding declaration.

6.5 Partition ELF format

A *partition image* contains all the information needed to “execute” the partition. It does not have loading or booting information. It contains one *image header structure*, one or more *partition header structures*, as well as the code and data that will be executed.

Since multiple partition headers are an experimental feature (to support multiprocessor in a partition), we will assume in what follows that a partition file contains only one image header structure and one partition header structure.

All the addresses of a partition image are absolute addresses which refer to the target RAM memory locations.

6.5.1 Partition image header

The partition image header is a data structure with the following fields:

```
struct xmImageHdr {
#define XMEF_PARTITION_MAGIC 0x24584d69 // $XM
    xm_u32_t sSignature;
    xm_u32_t compilationXmAbiVersion; // XM's abi version
    xm_u32_t compilationXmApiVersion; // XM's api version
    xm_u32_t noCustomFiles;
    struct xefCustomFile customFileTab[CONFIG_MAX_NO_CUSTOMFILES];
    xm_u32_t eSignature;
}
```

```
|} __PACKED;
```

1595

Listing 6.5: core/include/xmef.h

sSignature and eSignature: Holds the start and end signatures that identify the structure as a XtratuM partition image.

compilationXmAbiVersion: XtratuM ABI version used to compile the partition. That is, the ABI version of the libxm and other accompanying utilities used to build the XEF file.

compilationXmApiVersion: XtratuM API version used to compile the partition. That is, the API version of the libxm and other accompanying utilities used to build the XEF file.

1600

The current values of these fields are:

```
#define XM_ABI_VERSION 3
#define XM_ABI_SUBVERSION 1
#define XM_ABI_REVISION 0

#define XM_API_VERSION 3
#define XM_API_SUBVERSION 1
#define XM_API_REVISION 2
```

1605

1610

Listing 6.6: core/include/hypercalls.h

Note that these values may be different to the API and ABI versions of the running XtratuM. This information is used by XtratuM to check that the partition image is compatible.

noCustomFiles: The number of extra files accompanying the image. If the image were Linux, then one of the modules would be the *initrd* image. Up to `CONFIG_MAX_NO_FILES` can be attached. The `moduleTab` table contains the locations in the RAM's address space of the partition where the modules shall be copied (if any). See section 5.17.

1615

customFileTab: Table information about the customisation files.

```
struct xefCustomFile {
    xmAddress_t sAddr;
    xmSize_t size;
} __PACKED;
```

1620

Listing 6.7: core/include/xmef.h

sAddr: Address where the customisation file shall be loaded.

1625

size: Size of the customisation file.

The address where the custom files are loaded shall belong to the partition.

The `xmImageHdr` structure has to be placed in a section named “.xmImageHdr”. An example of how the header of a partition can be created is shown in section ??.



The remainder of the image is free to the partition developer. There is not a predefined format or structure of where the code and data sections shall be placed.

1630

6.5.2 Partition control table (PCT)

In order to minimize the overhead of the para-virtualised services, XtratuM defines a special data structure which is shared between the hypervisor and the partition called *Partition control table* (PCT). There is a PCT for each partition. XtratuM uses the PCT to send relevant operating information to the partitions. The PCT is mapped as read-only, allowing a partition only to read it. Any write access causes a system exception.

1635

```

typedef struct {
    xm_u32_t magic;
    xm_u32_t xmVersion; // XM version
1640    xm_u32_t xmAbiVersion; // XM's abi version
    xm_u32_t xmApiVersion; // XM's api version
    xm_u32_t resetCounter;
    xm_u32_t resetStatus;
    xm_u32_t cpuKhz;
1645 #define PCT_GET_PARTITION_ID(pct) ((pct)->id&0xff)
#define PCT_GET_VCPU_ID(pct) ((pct)->id>>8)
    xmId_t id;
    xmId_t noVCpus;
#ifdef CONFIG_ARM
1650    xm_u32_t irqIndex; // irq number raised
#endif /*CONFIG_ARM*/
    // Copy of kthread->ctrl.flags
    xm_u32_t flags;
1655 #define PARTITION_SYSTEM_F (1<<0) // 1:SYSTEM
#define PARTITION_FP_F (1<<1) // Floating point enabled
#define PARTITION_HALTED_F (1<<2) // 1:HALTED
#define PARTITION_SUSPENDED_F (1<<3) // 1:SUSPENDED
#define PARTITION_READY_F (1<<4) // 1:READY
//#define PARTITION_FLUSH_DCACHE_F (1<<5)
//#define PARTITION_FLUSH_ICACHE_F (1<<6)
1660 #define PARTITION_DCACHE_ENABLED_F (1<<7)
#define PARTITION_ICACHE_ENABLED_F (1<<8)
    xm_u32_t imgStart;
    xm_u32_t hwIrqs[HWIRQS_VECTOR_SIZE]; // Hw interrupts belonging to the
1665 partition
    xm_s32_t noPhysicalMemAreas;
    xm_s32_t noCommPorts;
    xm_u8_t name[CONFIG_ID_STRING_LENGTH];
    xm_u32_t hwIrqsPend[HWIRQS_VECTOR_SIZE]; // pending hw irqs
1670    xm_u32_t hwIrqsMask[HWIRQS_VECTOR_SIZE]; // masked hw irqs
    xm_u32_t extIrqsPend; // pending extended irqs
    xm_u32_t extIrqsMask; // masked extended irqs
    struct pctArch arch;
    struct {
1675        xm_u32_t noSlot:16, releasePoint:1, reserved:15;
        xm_u32_t id;
        xm_u32_t slotDuration;
    } schedInfo;
    xm_u16_t trap2Vector[NO_TRAPS];
1680    xm_u16_t hwIrq2Vector[CONFIG_NO_HWIRQS];
    xm_u16_t extIrq2Vector[XM_VT_EXT_MAX];
} partitionControlTable_t;

```

Listing 6.8: core/include/guest.h

The libxm call `XM_params.get_PCT()` returns a pointer to the PCT.

The architecture dependent part is defined in:

```

struct pctArch {
    xmAddress_t tbr;
1690 #ifdef CONFIG_MMU
    //#define _ARCH_PTDL1_REG ptdL1S
    // volatile xm_u32_t faultStatusReg;
    // volatile xm_u32_t faultAddressReg;

```



```
#endif /* CONFIG_MMU */
    xm_u32_t devices;
};
```

1695

Listing 6.9: core/include/arm/guest.h

signature: Signature to identity this data structure as a PCT.

xmAbiVersion: The ABI version of the currently running XtratuM. This value is filled by the running XtratuM.

xmApiVersion: The API version of the currently running XtratuM. This value is filled by the running XtratuM.

resetCounter: A counter of the number of partition resets. This counter is incremented when the partition is WARM reset. On a COLD reset it is set to zero. 1700

resetStatus: If the partition had been reset by a `XM_reset_partition()` hypercall, then the value of the parameter `status` is copied in this field. Zero otherwise.

id: The identifier of the partition. It is the unique number, specified in the `XM.CF` file, to unequivocally identify a partition. 1705

irqIndex: If an interruption is raised, then the irq number contained in the register `IRQINDEX` is copied in this field. `0xFFFFFFFF` otherwise.

fsr: If an Data Abort or Prefetch Abort exceptions are raised, then the Data or Instruction Fault Status contained in the registers `DFSR` or `IFSR` respectively are copied in this field. `0xFFFFFFFF` otherwise.

far: If an Data Abort or Prefetch Abort exceptions are raised, then the Data or Instruction Fault Address contained in the registers `DFAR` or `IFAR` respectively are copied in this field. `0xFFFFFFFF` otherwise. 1710

hwIrqs: A bitmap of the hardware interrupts allocated to the partition. Hardware interrupts are allocated to the partition in the `XM.CF` file.

noPhysicalMemoryAreas: The number of memory areas allocated to the partition. This value defines the size of the `physicalMemoryAreas` array. 1715

name: Name of the partition.

hwIrqsPend: Bitmap of the hardware interrupts allocated to the partition delivered to the partition. ¹

extIrqsPend: Bitmap of the extended interrupts allocated to the partition delivered to the partition.

hwIrqsMask: Hardware interrupts masked for the partition.. ²

extIrqsMask: Extended interrupts masked for the partition. 1720

In the current version, the `irqIndex`, `fsr` and `far` fields are specific of the ARM architecture.

6.6 XEF format

The XEF is a wrapper for the files that may be deployed in the target system. There are three kinds of files:

- Partition images.
- The XtratuM image.
- Customisation files.

A XEF file has a header (see listing 6.6) and a set of *segments*. Each segment, like in ELF, represents a block of memory that shall be loaded into RAM. 1725

The tool `xmeformat` converts from ELF or plain data files to XEF format, see chapter 9.

¹This field is maintain for compatibility, but not used.

²This field is maintain for compatibility, but not used.

```

1730 struct xefHdr {
    #define XEF_SIGNATURE 0x24584546
        xm_u32_t signature;
        xm_u32_t version;
    #define XEF_DIGEST 0x1
    #define XEF_COMPRESSED 0x4
1735 #define XEF_RELOCATABLE 0x10

    #define XEF_TYPE_MASK 0xc0
    #define XEF_TYPE_HYPERVISOR 0x00
    #define XEF_TYPE_PARTITION 0x40
1740 #define XEF_TYPE_CUSTOMFILE 0x80

    #define XEF_ARCH_SPARCv8 0x400
    #define XEF_ARCH_MASK 0xff00
        xm_u32_t flags;
1745        xm_u8_t digest[XM_DIGEST_BYTES];
        xm_u8_t payload[XM_PAYLOAD_BYTES];
        xmSize_t fileSize;
        xmAddress_t segmentTabOffset;
        xm_s32_t noSegments;
1750        xmAddress_t customFileTabOffset;
        xm_s32_t noCustomFiles;
        xmAddress_t imageOffset;
        xmSize_t imageLength;
        xmSize_t deflatedImageLength;
1755        xmAddress_t pageTable;
        xmSize_t pageTableSize;
        xmAddress_t xmImageHdr;
        xmAddress_t entryPoint;
1760 } __PACKED;

```

Listing 6.10: core/include/xmef.h

signature: A 4 bytes word to identify the file as a XEF format.

version: Version of the XEF format.

flags: Bitmap of features present in the XEF image. It is a 4 bytes word. The existing flags are:

XEF_DIGEST: If set, then the `digest` field is valid and shall be used to check the integrity of the XEF file.

XEF_COMPRESSED: If set, then the partition binary image is compressed.

XEF.CIPHERED: (**future extension**) to inform whether the partition binary is encrypted or not.

XEF.CONTENT: Specifies what kind of file is.

digest: when the `XEF_DIGEST` flag is set, this field holds the result of processing whole the XEF file (supposing the digest field set to 0). The MD5 algorithm is used to calculate this field.

Despite the well known security flaws, we selected the MD5 digest algorithm because it has a reasonable trade-off between calculation time and security level. . Note that the digest field is used to detect accidental modifications rather than intentional attacks. In this scenario, the MD5 is a good choice.

payload: This field holds 16 bytes which can be freely used by the partition supplier. It could be used to hold information such as partition's version, etc.

The content of this field is never modified by XtratuM.

fileSize: XEF file size in bytes.

segmentTabOffset: Offset to the section table.

noSegments: Number of segments held in the XEF file. In the case of a customisation file, there will be only one segment.

customFileTabOffset: Offset to the custom files table. 1780

noCustomFiles: Number of custom files.

imageOffset: Offset to the partition binary image.

imageLength: Size of the partition binary image.

deflatedImageLength: When the XEF_COMPRESS flag is set, this field holds the size of the uncompressed partition binary image. 1785

xmImageHdr: Pointer to the partition image header structure (`xmImageHdr`). The `xmeformat` tool copies the address of the corresponding section in this file.

entryPoint: Address of the starting function.

Additionally, analogously to the ELF format, XEF contemplates the concept of *segment*, which is, a portion of code/data with a size and a specific load address. A XEF file includes a segment table (see listing 6.6) which describes each one of the sections of the image (custom data XEF files have only one section). 1790

```

struct xefSegment {
    xmAddress_t physAddr;
    xmAddress_t virtAddr;
    xmSize_t fileSize;
    xmSize_t deflatedFileSize;
    xmAddress_t offset;
} __PACKED;

```

Listing 6.11: `core/include/xmef.h`

startAddr: Address where the segment shall be located while it is being executed. This address is the one used by the linker to locate the image. If there is not MMU, then `physAddress=virtAddr`.

fileSize: The size of the segment within the file. This size could be different regarding the memory required to be executed (for example a data segment (.bss segment) usually requires more memory once loaded into memory). 1795

deflatedFileSize: When the XEF_COMPRESS flag is set, this field holds the size of the segment when uncompressed. 1800

offset: Location of the segment expressed as an offset in the partition binary image. 1805

6.6.1 Compression algorithm

The compression algorithm implemented is Lempel-Ziv-Storer-Szymanski (LZSS). It is a derivative of LZ77, that was created in 1982 by James Storer and Thomas Szymanski. A detailed description of the algorithm appeared in the article “Data compression via textual substitution” published in Journal of the ACM. 1810

The main features of the LZSS are:

1. Fairly acceptable trade-off between compression rate and decompression speed.
2. Implementation simplicity.
3. Patent-free technology. 1815

Aside from LZSS, other algorithms which were regarded were: huffman coding, gzip, bzip2, LZ77, RLE and several combinations of them. Table 6.2 sketches the results of compressing XtratuM’s core binary with some of these compression algorithms.

6.7 Container format

A *container* is a file which contains a set of XEF files.

Algorithm	Compressed size	Compression rate (%)
LZ77	43754	44.20%
LZSS	36880	53.01%
Huffman	59808	23.80%
Rice 32bits	78421	0.10%
RLE	74859	4.60%
Shannon-Fano	60358	23.10%
LZ77/Huffman	36296	53.76%

Table 6.2: Outcomes of compressing the `xm_core.bin` (78480 bytes) file.

The tool `xmpack` manages container files (see chapter 9).

A *component* is an executable binary (hypervisor or partition) jointly with associated data (configuration or customization file). The XtratuM component contains the files: `xm_core.bin` and `XM.CT.bin`. A partition component is formed by the partition binary file and zero or more customization files.

XtratuM is not a boot loader. There shall be an external utility (the resident software or boot loader) which is in charge of coping the code and data of XtratuM and the partition from a permanent memory into the RAM. Therefore, the the container file is not managed by XtratuM but by the resident software, see chapter 7.

Note also, that the container does not have information regarding where the components shall be loaded into RAM memory. This information is contained in the header of the binary image of each component.

The container file is like a packed filesystem which contains file metadata (name of the files) along with the contents of each file. Also, the file which contains the executable image and the customisation data of each partition is specified.

The container holds the following elements:

1. The header (`xmefContainerHdr` structure). A data structure which holds pointers (in the form of offsets) and the sizes to the remaining sections of the file.
2. The component table section, which contains an array of `xmefComponent` structures. Each element contains information about one component.
3. The file table section, which contains an array of files (`xmefFile` structure) in the container.
4. The string table section. Contains the names of the files of the original executable objects. This is currently used for debugging.
5. The file data table section, with the actual data of the executable (XtratuM and partition images) and the configuration files.

The container header has the following fields:

```

struct xmefContainerHdr {
    xm_u32_t signature;
#define XM_PACKAGE_SIGNATURE 0x24584354 // $XCT
    xm_u32_t version;
#define XMPACK_VERSION 3
#define XMPACK_SUBVERSION 0
#define XMPACK_REVISION 0
    xm_u32_t flags;
#define XMEF_CONTAINER_DIGEST 0x1
    xm_u8_t digest[XM_DIGEST_BYTES];
    xm_u32_t fileSize;
    xmAddress_t partitionTabOffset;
    xm_s32_t noPartitions;
    xmAddress_t fileTabOffset;
    xm_s32_t noFiles;
    xmAddress_t strTabOffset;
    xm_s32_t strLen;

```

```

    xmAddress_t fileDataOffset;
    xmSize_t fileDataLen;
} __PACKED;

```

Listing 6.12: core/include/xmef.h

signature: Signature field. 1865

version: Version of the package format.

flags:

digest: Not used. Currently the value is zero.

fileSize: The size of the container.

partitionTabOffset: The offset (relative to the start of the file) to the partition array section. 1870

noPartitions: Number of partitions plus one (XtratuM is also a component) in the container.

componentOffset: The offset (relative to the start of the file) to the component's array section.

fileTabOffset: The offset (relative to the start of the container file) to the files's array section.

noFiles: Number of files (XtratuM core, the XM.CT file, partition binaries, and partition-customization files) in the container. 1875

strTabOffset The offset (relative to the start of the container file) to the strings table.

strLen The length of the strings table. This section contains all the names of the files.

fileDataOffset The offset (relative to the start of the container file) to the file data section.

fileDataLen The length of the file data section. This section contains all the contents of all the components.

Each entry of the partition table section describes all the XEF files that are part of each partition. It contains the following fields: 1880

```

struct xmefPartition {
    xm_s32_t rId;
    xm_s32_t file;
    xm_u32_t noCustomFiles;
    xm_s32_t customFileTab[CONFIG_MAX_NO_CUSTOMFILES];
} __PACKED;

```

Listing 6.13: core/include/xmef.h

id: The identifier of the partition. 1890

file: The index into the file table section of the XEF partition image.

noCustomFiles: Number of customisation files of this component.

customFileTab: List of custom file indexes.

The metadata of each file is store in the file table section:

```

struct xmefFile {
    xmAddress_t offset;
    xmSize_t size;
    xmAddress_t nameOffset;
} __PACKED;

```

Listing 6.14: core/include/xmef.h

offset: The offset (relative to the start of the file data table section) to the data of this file in the container.

size: The size reserved to store this file. It is possible to define the size reserved in the container to store a file independently of the actual size of the file. See the section 9.3.1 tool.

nameOffset: Offset, relative to the start of the strings table, of the name of the file. 1905

The strings table contains the list of all the file names.

The file data section contains the data (with padding if `fileSize <= size`) of the files.

This page is intentionally left blank.

Chapter 7

Booting

Has been developed a small booting code called *resident software*, which is in charge of the initial steps of booting the processor.

In this ARM CORTEX-A9, the primary processor starts up the system, as in the mono-core version. Once this sequence is finished, this processor initialises secondary processors and waits in a synchronisation point to start partition's scheduling.

The *resident software* is in charge of loading into memory XtratuM, its configuration file (XM_CT) and any partition jointly with its customisation file as found in the container. The information hold by the XM_CT file is used to load any partition image. Additionally, the *resident software* informs XtratuM which partitions have to be booted.

After starting, XtratuM assumes that the partitions informed as ready-to-be-booted are in RAM/SRAM memory, setting them in running state right after it finishes its booting sequence

If a partition's code is not located within the container, then XtratuM sets the partition in HALT state until a system partition resets it by using `XM_reset_partition()` hypercall. In this case, the RAM image of the partition shall be loaded by a system partition through the `XM_memory_copy()` hypercall.

Note that there may be several booting partitions. All those partitions will be started automatically at boot time.

The boot process will consist in the initialisation of the required devices for the whole system to operate, and in the initialisation of the hypervisor itself. XtratuM will take full control of the process.

In the standard boot procedure of the ARM CORTEX-A9 processor, the program counter register is initialized with address of the *Reset exception* handler and the processor mode is set to *Supervisor* with interrupts disabled.

The boot sequence of XtratuM in a ARM CORTEX-A9 architecture can be summarised as:

1. When the processor is started (reset) the resident software is executed by the processor 0. This is a small code that performs the following actions:
 - (a) Initializes a stack (required to execute "C" code).
 - (b) Installs a exception handler table (only for the case that its code would generate a fault, XtratuM installs a new exceptions handler during its initialisation).
 - (c) Checks that the data following the resident software in the memory is a container (a valid signature), and seeks the XtratuM hypervisor through container (a valid signature).
 - (d) Copies the XtratuM hypervisor and booting partitions into RAM memory.
 - (e) Jumps to the entry point of the hypervisor in RAM memory.
2. Installs the XtratuM exception handler table.
3. Set a valid C environment: zero the BSS, zero the general processor registers and load a valid stack, and copy data sections.
4. Hypervisor initialisation:
 - (a) Initialise the internal console.



- (b) Detect the processor frequency (information extracted from the XML configuration file).
- (c) Initialise memory manager.
- (d) Wake up the secondary CPUs.
- (e) Initializes the interrupt controller.
- 1945 (f) Initialise hardware and virtual timers.
- (g) Initialise the scheduler.
- (h) Initialise the communication channels.
- (i) For each partition, set their stored PC to point to their reset exception handler.
- (j) Finally, calling the scheduler and becoming into the idle task.

7.1 Partition booting

1950 The partitions boot the same way they would do in a native ARM CORTEX-A9 environment, that is, a *Reset exception* is raised for them.

Take into account that partitions will start their real execution once their first execution slot has been reached and they can handle the generated virtual reset exception.

Chapter 8

Configuration

This section describes how XtratuM is configured. There are two levels of configuration. The first level is concerned with source code configuration in order to customise the resulting XtratuM executable image. Since XtratuM does not use dynamic memory to setup internal data structures, most of these configuration parameters are related to the size, or ranges, of the statically created data structures (maximum number of partitions, channels, etc..).

The second level of configuration is done via an XML file. This file configures the resources allocated to each partition.

8.1 XtratuM source code configuration (menuconfig)

There are two different blocks that shall be configured: 1) XtratuM source code; and 2) the Xtratum Abstraction Layer (XAL). The configuration menu of each block is presented one after the other when executed the “\$ make menuconfig” command in the root directory of XtratuM sources.

The next table lists all the XtratuM configuration options and its default values. Note that since there are logical dependencies between some options, the menuconfig tool may not show all the options.

Parameter	Type	Possible values
Processor		
ARM cpu	choice	[Cortex-A9]
Board	choice	[ZEDBOARD] [ZYNQ ZC706] [Imperas Sim-OVPSIM]
Simulated Board	choice	[ZEDBOARD] [ZC706]
Multicore support	choice	[None] [SMP support]
Number of CPUs supported	int	2 if (ZYNQ)
Enable L2 cache	bool	y
Select L2 cache policy	choice	[Write-back/write-allocate] [Write-through/no write-allocate] [Write-back/no write-allocate]
Physical memory layout		
XM load address	hex	0x20000000
XM load data address	hex	0x21000000
XM virtual address	hex	0x20000000
Enable experimental features	bool	n
<i>Continues...</i>		

Parameter	Type	Possible values
Enable assertions	bool y	
Debug and profiling support	bool y	
Dump CPU state when a trap is raised	bool y	
Verbose hypercalls execution	bool n	
Max. identifier length (B)	int 16	
Hypervisor		
Enable voluntary preemption support	bool n	
Kernel stack size (KB)	int 8	
Number of virtual CPUs	int 1	
Number of IPVIs	int 4	
Enable external synchronisation	bool n	
Enable kernel audit events	bool n	
Select XM Data section cache policy	choice	[Write-back/write-allocate] [Uncacheable]
Select XM Code section cache policy	choice	[Write-back/write-allocate] [Write-through/no write-allocate] [Write-back/no write-allocate]
Select L1 cache policy for partitions	choice	[Write-back/write-allocate] [Write-through/no write-allocate] [Write-back/no write-allocate]
MMU		
Drivers		
Enable UART Driver	bool y	
Reserve UART0	bool n	
UART 0 Reference Clock (Hz) - XPAR_PS7_UART_0_UART_CLK_FREQ_HZ	int 50000000	
Reserve UART1	bool n	
UART 1 Reference Clock (Hz) - XPAR_PS7_UART_1_UART_CLK_FREQ_HZ	int 50000000	
Enable early output	bool n	
Select early UART port	choice	[UART0] [UART1]
Early UART baudrate	int 115200	
Enable the addition of a carriage return at the end of the line.	bool n	
Enable memory block driver	bool y	
Objects		
Verbose HM events	bool y	
Enable XM/partition status accounting	bool n	
Size of the XM's internal buffer to store trace logs	int 20	
Size of the XM's internal buffer to store hm logs	int 20	

1965 **ARM cpu:** Processor model.

Board: Enables the specific board features: memory protection units, timers, UART and interrupt controller.

Simulated Board: In case of use a simulator instead a real board.

Multicore support: SMP operative is allowed.

Number of CPUs supported: Number of processor cores.

1970 **Enable L2 cache:** This option enables the L2 cache controller (L2Cpl310).

L2 cache policy: This option defines the policy of the L2 cache.

XtratuM load address: Physical RAM address where XtratuM resides. This value shall be the same than the one specified in the XM_CF file.

1975 **XtratuM data load address:** Physical RAM address where XtratuM data resides. This value shall be the same than the one specified in the XM_CF file.

XtratuM virtual address: Virtual address where XtratuM resides once the MMU is enabled.

Enable experimental features: Enable this option to be able to select experimental features. This option does not do anything by itself, just shows the options marked as experimental if exists.

Debug and profiling support: XtratuM is compiled with debugging information (gcc flag “-ggdb”) and assert code is included. This option should be used only during the development of the XtratuM hypervisor. 1980

Dump CPU state when a trap is raised: When enabled, after a trap, XtratuM shows the state of the processor.

Verbose hypercalls execution: XtratuM prints extra information about Hypercalls execution.

Maximum identifier length (B): The maximum string length (including the terminating “0x0” character) of the names: partition name, port name, plan, etc. Since the names are only used for debugging, 16 characters is a fair number. 1985

Kernel stack size (KB): Size of the stack allocated to each partition. It is the stack used by XtratuM when attending the partition hypercalls.

Do not change (reduce) this value unless you know what you are doing.

Enable UART driver: Enables UART port support.

XM Data section cache policy: The policy of the L1 cache for XtratuM Data section. 1990

XM Code section cache policy: The policy of the L1 cache for XtratuM Code section.

L1 cache policy for partitions: The policy of the L1 cache for partitions.

UART X Reference Clock: Input clock used by the UART - XPAR_PS7_UART_X.UART_CLK_FREQ_HZ.

The valid values are 50MHz and 100Mhz. These frequencies allow to configure three baudrates:9600, 115200, 230400. 1995

Enable early output: By default, the first booting messages of XtratuM are not displayed until the output device is started. Enable this option if you want to see even those initial messages.

Early UART baudrate: The bps is configured with this option. The UART bsp, when managed by XtratuM, is always set to 115200, regardless of this parameter.

Enable memory block driver: This option enables the support for memory blocks. These memory blocks are used to store the logs. 2000

Verbose HM events: If set, the occurrence of a HM event is always outputted on console.

Enable XM/partition status accounting: Enable this option to collect statistical information about XtratuM itself and about the partitions.

Note that this feature increases the overhead of most of the XtratuM operations. 2005

8.2 Hypervisor configuration file (XM_CF)

The XM_CF file defines the system resources, and how they are allocated to each partition.

For an exact specification of the syntax (mandatory/optional elements and attributes, and how many times an element can appear) the reader is referred to the XML schema definition in the Appendix A.

8.2.1 Data representation and XPath syntax

When representing physical units, the following syntax shall be used in the XML file:

Time: Pattern: “[0-9]+(.[0-9]+)?([mu]?[sS])” 2010

Examples of valid times:

```
9s      # nine seconds.
10ms    # ten milliseconds.
0.5ms   # zero point five milliseconds.
500us   # five hundred microseconds =0.5ms
```

2015

Size: Pattern: “[0-9]+(.[0-9]+)?([MK]?B)”

Examples of valid sizes:

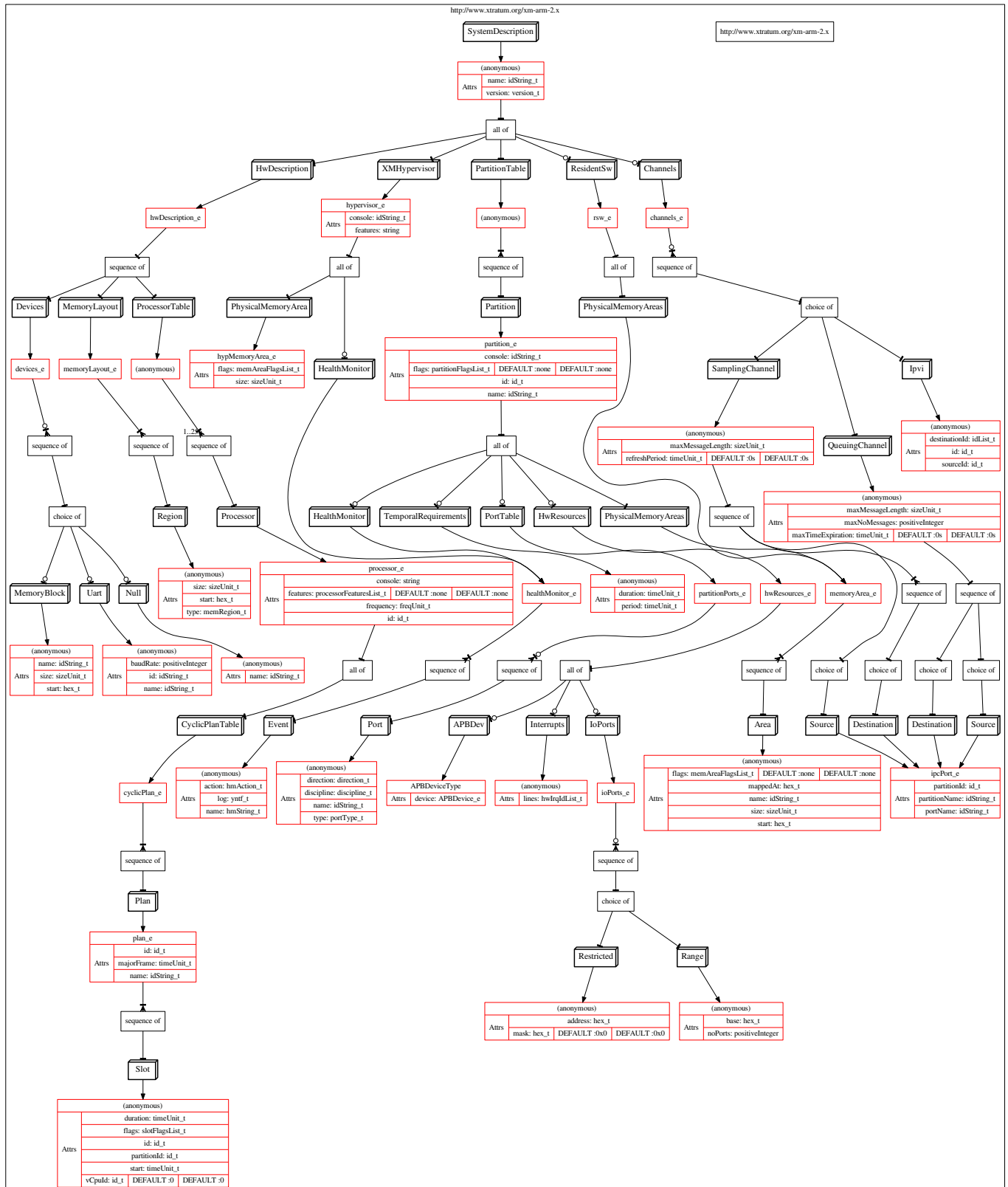


Figure 8.1: XML Schema diagram.

90B # ninety bytes.
 50KB # fifty Kilo bytes =(50*1024) bytes.
 2MB # two mega bytes =(2*1024*1024) bytes.
 2.5KB # two point five kilo bytes =2560B.

2020

It is advised not to use the decimal point on sizes.

Frequency: Pattern: “[0-9]+(.[0-9]+)?([MK][Hh]z)”

Examples of valid frequencies:

80Mhz # Eighty mega hertz = 80000000 hertz.
 20000Khz # Twenty mega hertz = 20000000 hertz.

2025

Boolean: Valid values are: “yes”, “true”, “no”, “false”.

Hexadecimal: Pattern: “0x[0-9a-fA-F]+”

Examples of valid numbers:

0xFfffFfff, 0x0, 0xF1, 0x80

2030

An XML file is organised as a set of nested elements, each element may contain attributes. The *XPath* syntax is used to refer to the objects (elements and attributes). Examples:

`/SystemDescription/PartitionTable` The element `PartitionTable` contained inside the element `SystemDescription`, which is the root element (the starting slash symbol).

`/SystemDescription/@name` Refers to the attribute `./@name` of the element `SystemDescription`.

2035

8.2.2 The root element: `/SystemDescription`

Figure 8.1 is a graphical representation of the schema of the XML configuration file. The types of the attributes are not represented, see the appendix A for the complete schema specification. An arrow ended with circle are optional elements.

The root element is “`/SystemDescription`”, which contain the mandatory `./@version`, `./@name` and `./@xmlns` attributes. The `xmlns` name space shall be “`http://www.xtratum.org/xm-arm-2.x`”.

2040

There are five second-level elements:

`/SystemDescription/XMHypervisor` Specifies the board resources (memory and processor plan) and the hypervisor health monitoring table.

`/SystemDescription/ResidentSw` This is an optional element providing information about the resident software.

2045

`/SystemDescription/PartitionTable` This is a container element which holds all the `./partition` elements.

`/SystemDescription/Channels` A sequence of channels which define port connections.

`/SystemDescription/HwDescription` Contains the configuration of physical and virtual resources.

8.2.3 The `/SystemDescription/XMHypervisor` element

There are two optional attribute `./@console` and `./@features`.

`./@console` Indicates the name of a device defined in the `/SystemDescription/HwDescription/Devices` section.

2050

`./@features` Indicates the hypervisor configurable features that will be enabled. **Note:** To enable more than one feature the attribute should be a list of names separated by spaces. As example: “`XM_HYP_FEAT.... XM_HYP_FEAT....`”.

Mandatory elements:

2055

`./PhysicalMemoryAreas` Sequence of memory areas allocated to XtratuM.

Optional elements:

./HealthMonitoring Contains a sequence of health monitoring event elements.

Not all HM actions can be associated with all HM events. Consult the allowed actions in the “Volume 4: Reference Manual”.

A health monitoring event element contains the following attributes:

./event/@name The event’s name. Below is the list of available events:

```
<xs:simpleType name="hmString_t">
  <xs:restriction base="xs:string">
    <xs:enumeration value="XM_HM_EV_INTERNAL_ERROR"/>
    <xs:enumeration value="XM_HM_EV_UNEXPECTED_TRAP"/>
    <xs:enumeration value="XM_HM_EV_PARTITION_ERROR"/>
    <xs:enumeration value="XM_HM_EV_PARTITION_INTEGRITY"/>
    <xs:enumeration value="XM_HM_EV_MEM_PROTECTION"/>
    <xs:enumeration value="XM_HM_EV_OVERRUN"/>
    <xs:enumeration value="XM_HM_EV_SCHED_ERROR"/>
    <xs:enumeration value="XM_HM_EV_WATCHDOG_TIMER"/>
    <xs:enumeration value="XM_HM_EV_INCOMPATIBLE_INTERFACE"/>
    #ifdef CONFIG_ARM
    <xs:enumeration value="XM_HM_EV_ARM_UNDEF_INSTR"/>
    <xs:enumeration value="XM_HM_EV_ARM_PREFETCH_ABORT"/>
    <xs:enumeration value="XM_HM_EV_ARM_DATA_ABORT"/>
    <xs:enumeration value="XM_HM_EV_ARM_DATA_ALIGNMENT_FAULT"/>
    <xs:enumeration value="XM_HM_EV_ARM_DATA_BACKGROUND_FAULT"/>
    <xs:enumeration value="XM_HM_EV_ARM_DATA_PERMISSION_FAULT"/>
    <xs:enumeration value="XM_HM_EV_ARM_INSTR_ALIGNMENT_FAULT"/>
    <xs:enumeration value="XM_HM_EV_ARM_INSTR_BACKGROUND_FAULT"/>
    <xs:enumeration value="XM_HM_EV_ARM_INSTR_PERMISSION_FAULT"/>
    #endif
  </xs:restriction>
</xs:simpleType>
```

Listing 8.1: user/tools/xmcparser/xmc.xsd.in

./event/@action The name of the action associated with this event. Below in the list of available actions:

```
<xs:simpleType name="hmAction_t">
  <xs:restriction base="xs:string">
    <xs:enumeration value="XM_HM_AC_IGNORE"/>
    <xs:enumeration value="XM_HM_AC_SHUTDOWN"/>
    <xs:enumeration value="XM_HM_AC_PARTITION_COLD_RESET"/>
    <xs:enumeration value="XM_HM_AC_PARTITION_WARM_RESET"/>
    <xs:enumeration value="XM_HM_AC_HYPERVISOR_COLD_RESET"/>
    <xs:enumeration value="XM_HM_AC_HYPERVISOR_WARM_RESET"/>
    <xs:enumeration value="XM_HM_AC_SUSPEND"/>
    <xs:enumeration value="XM_HM_AC_HALT"/>
    <xs:enumeration value="XM_HM_AC_PROPAGATE"/>
    <xs:enumeration value="XM_HM_AC_SWITCH_TO_MAINTENANCE" />
  </xs:restriction>
</xs:simpleType>
```

Listing 8.2: user/tools/xmcparser/xmc.xsd.in

./event/@log Boolean flag to select whether the event will be logged or not.

8.2.4 The /SystemDescription/HwDescription element

It contains three mandatory elements:

./HwDescription/ProcessorTable Which holds a sequence of **./Processor** elements. Each processor element describes one physical processor: the processor clock **./@frequency** (the frequency units has to be specified.

The timer and the clock are derived from this value. The clock frequencies shall be multiple of 50MHz in order to avoid an error in the prescaler computed value.), `./@id` (zero in a mono-processor system), and an optional `./@features` attribute. The `./@features` attribute contains a list of specific processor features than can be selected. Currently, only the memory protection workaround (“XM.CPU_LEON2_WA1”), for the memory mapped processor registers bug¹. The `./@console` attribute enables to use a console when in supervisor mode for all the output generated by this processor. The attribute is optional, overriding the `./Hypervisor/@console` attribute. 2110 2115

Also, the `./ProcessorTable/Processor` element defines the scheduling plan of this processor. It is specified in the element `./Processor/Sched/CyclicPlan/Plan`². The `./Plan` element has the required attributes `./@name` and `./majorFrame`. It also contains the sequence of `./Slots` that form the plan.

Each `./Slot` element has the following attributes: 2120

`./Slot/@id` Slot ids shall meet the ids’ rules defined in section 2.4. This value can be retrieved by the partition at run time (see section 5.5.1).

`./Slot/@duration` Time duration of the slot.

`./Slot/@partitionId` Id of the partition that will be executed during this slot.

`./Slot/@start` Offset with respect to the MAF start. 2125

`./Slot/@vCpuId` Id of the partition virtual CPU that will execute this slot.

`./Slot/@flags` Flags applicable to this slot. Valid values are: “periodStart”.

Slots intervals shall not overlap.

`./HwDescription/MemoryLayout` Defines the memory layout of the board. All the memory allocated to partitions, resident software and XtratuM itself shall be in the range of one of these areas. 2130

`./HwDescription/Devices` The device element contains the sequence of XtratuM devices. Currently XtratuM implements two types of devices: UART and memory blocks.

`./Uart` Has the required attributes `./Uart/@name`, `./Uart/@baudRate` and `./Uart/@id`. This element associates the hardware device `@id` with the `@name`, and programs the transmission speed.

`./MemoryBlockTable` This element contains a sequence of one or more `./Block` elements. A memory block device defines an area of RAM (ROM or FLASH) memory. This block of memory can then be used to store traces, health monitoring logs or the console output of a partition. Below is the list of attributes of the `./Block` element: 2135

`./MemoryBlockTable/Block/@name` Required. Name which identifies the device. This name is only used to refer to this device in the configuration file. Once compiled the configuration file this name is removed. 2140

`./MemoryBlockTable/Block/@start` Required. Starting address of the memory block.

`./MemoryBlockTable/Block/@size` Required. Size of the memory block.

8.2.5 The `/SystemDescription/ResidentSw` element

The element `./PhysicalMemoryAreas` is used to declare the memory areas where the resident software will be located. This information is included in the configuration file for completeness (all the memory areas of the board shall be described in the configuration file) and used only to check memory overlap errors. 2145

Also the attribute `./@entryPoint` is used by XtratuM in the case of a cold system reset. In that case, XtratuM will give back the control of the system to the resident software by jumping to this address.

8.2.6 The `/SystemDescription/PartitionTable/Partition` element

Attribute description:

¹Some key processor registers, needed to guarantee the spatial isolation, are mapped memory addresses which are not monitored/protected by the write protection mechanism. The workaround consists in protecting this register area using the watchpoint mechanism. The workaround is only applicable if the watchpoint facility is present.

²The large number of nested elements is for future compatibility with multiple plans and scheduling policies.

2150

./@id Required. See section 2.4 for a description on how to identify XtratuM objects.

./@name Optional.

./@console Optional. The console device where the output of the hypercall `XM_write_console()` is copied to.

./@flags Optional. List of features. Possible values are:

fp If set, the partition is allowed to use floating point operations. Not set by default.

2155

boot If set, then the XtratuM will set this partition in running state after a XtratuM reset. The resident software shall load in RAM the image of this partition.

icache_disabled If set icache is disabled while the partition is being executed.

dcache_disabled If set dcache is disabled while the partition is being executed.

system If set, the partition has system privileges. Not set by default.

2160

Partition elements:

./PhysicalMemoryAreas Sequence of memory areas allocated to the partition.

./HwResources Contains the list of interrupts and IO ports allocated to the partition.

./PortTable Contains the sequence of communication ports (queuing and sampling ports) of the partition.

2165

./Trace Configuration of the trace facility of the partition. Same attributes than that of the `/SystemDescription/XMHypervisor` element.

./TemporalRequirements An element which has two mandatory attributes: **./@period** and **./@duration**.
This data is not checked by XtratuM. Reserved for future use.

Configuration of memory areas

The attributes are **@start**, **@size** and **@flags**. The **@flags** attribute is a list of the following values:

Value	Description
unmapped	Allocated to the partition, but not mapped by XtratuM in the page table.
mappedAt	It allows to allocate this area to a virtual address.
read-only	The area is write-protected to the partition.
uncacheable	Memory cache is disabled.

Configuration of I/O ports

2170

There are two ways to allocate a port to a partition: using ranges of ports, and using the restricted port allocation. Both are declared by elements contained in the **./Partition/HwResources/IOPorts** element:

./Range A range of port addresses is allocated to the partition. The attributes of a range element are:

./Range/@base Required hexadecimal base address.

./Range/@noPorts Required number of ports in this range. Each port is a word (4 bytes).

2175

./Restricted An I/O port which is partially controlled by the partition. The attributes are:

./Restricted/@address Required hexadecimal address of the port.

./Restricted/@mask Optional (4 bytes hexadecimal). The bits set in this mask can be read and written by the partition.

2180

Those bits not allocated to this partition (i.e. the bit not set in the bitmask) can be allocated to other partitions.

Configuration of APB Devices

Devices are accessed by the user directly in bare metal without hypervisor intervention. Even so, XtratuM manage which devices are accessible and which are protected or masked. Therefore, if a partition should make use of a device, it must be assigned in the configuration file using the attribute `./@device` in the element `./Partition/HwResources/APBDev`.

Note: If a partition should access more than one device, the attribute should be a list of names separated by spaces. As example: “ZYNQ_DEV_UA0 ZYNQ_DEV_SPI0”. 2185

Note: If any of the UART devices were reserved for XtratuM in the source code configuration process(8.1), would not be available in this list.

Device	Tag Name	Base Address
UART 0	ZYNQ_DEV_UA0	0xE0000000
UART 1	ZYNQ_DEV_UA1	0xE0001000
USB 0	ZYNQ_DEV_USB0	0xE0002000
USB 1	ZYNQ_DEV_USB1	0xE0003000
I2C 0	ZYNQ_DEV_I2C0	0xE0004000
I2C 1	ZYNQ_DEV_I2C1	0xE0005000
SPI 0	ZYNQ_DEV_SPI0	0xE0006000
SPI 1	ZYNQ_DEV_SPI1	0xE0007000
CAN 0	ZYNQ_DEV_CAN0	0xE0008000
CAN 1	ZYNQ_DEV_CAN1	0xE0009000
GPIO	ZYNQ_DEV_GPIO	0xE000A000
GEM 0	ZYNQ_DEV_GEM0	0xE000B000
GEM 1	ZYNQ_DEV_GEM1	0xE000C000
QSPI	ZYNQ_DEV_QSPI	0xE000D000
SMC	ZYNQ_DEV_SMC	0xE000E000
TRIPLE TIMER	ZYNQ_DEV_TTC1	0xF8002000

Table 8.2: List of ZYNQ Device Names.

Configuration of interrupts

The element `./Partition/HwResources/Interrupts` has the attribute `./@lines` which is a list of the interrupt numbers (in the range 0 to 96) allocated to the partition. 2190

Note: Some interrupt lines (Table 8.3) are reserved for XtratuM as these are needed for basic functionalities of the hypervisor. Additionally system offers the possibility of emulate some of these functionalities using the extended interrupts(Table 5.4).

Interrupt Line	Device	Functionality	Comment
27	Global Timer	System Clock	
29	Private Timer	System Timer	
0	SPI 0	Generate Extended Interrupts	
59	UART 0	System Console	If reserved
82	UART 1	System Console	If reserved

Table 8.3: List Reserved Interrupt Lines.

8.2.7 The /SystemDescription/Channels element

This is an optional element with no attributes and which contains a list of channel elements. There are three types of channels:

./SamplingChannel Shall contain one **./Source** element or a **./ExternalSource** and one or more **./Destination** or **./ExternalDestination** elements. It has the following attributes:

./@maxLength Required. The maximum message size that can be stored on this channel.

./@refreshPeriod Optional. The duration of validity of a written message. When a message is read after this period, the validity flag will be false.

./@address Optional. Address where the channel must be located at.

./@size Optional. Maximum size allocated to the channel.

./QueuingChannel Shall contain one **./Source** or **./ExternalSource** element and one **./Destination** or **./ExternalDestination** element. It has the following attributes:

./@maxLength Required. The maximum message size that can be stored on this channel.

./@maxNoMessages Required. The maximum number of messages that will be stored in the channel.

./@address Optional. Address where the channel must be located at.

./@size Optional. Maximum size allocated to the channel.

./Ipvi It has the following attributes:

./@id Required. Identifier of the IPVI.

./@sourceId Required. Partition who raises the IPVI.

./@destinationId Required. Partition or list of partitions who receives the IPVI.

Note: In the IPVIs, the pair (**@id**, **@destinationId**) cannot be replicated, since each partition have just one IPVI **@id**. Therefore, it is acceptable to have several IPVIs **@id**=0, but with different **@destinationId** values.

Note: The **./QueuingChannel/@validPeriod** attribute has been removed with respect to XtratuM-2.2.x versions.

The arguments **maxNoMsgs** and **maxMsgSize** of the hypercalls **XM.create_queuing_port()** and **XM.create_sampling_port()** shall match the values of the attributes **./@maxNoMessages** and **./@maxNoMessages**.

The **./Source** and the **./Destination** elements have the following attributes:

./@partitionId Required. Id of the partition owning the port which is plugged to this channel.

./@partitionName Optional. Name of the partition owning the port which is plugged to this channel.

./@portName Required. Name of the port which is being binded with this channel.

The **./ExternalSource** and the **./ExternalDestination** elements have the following attributes:

./@portName Required. Name of the port which is being binded with this channel.

./@cpuId Required. Id of the CPU which is executing the hypervisor instance where this port exists.

The XML schema which defines the configuration file is in the appendix A.

Chapter 9

Tools

This section describes the tools to assist the integrator and the partition developers in the process of building the final system file.

xmcparser: System XML configuration parser.

xmeformat: Converts ELF files into XEF ones.

2230

elfbdr.py: Creates an ELF le with all the components of a XtratuM-based system.

xmboottab: The tool extracts the loading information for one or more partitions.

9.1 XML configuration parser (xmcparser)

The utility `xmcparser` translates the XML configuration file containing the system description into binary form that can be directly used by XtratuM.

In the first place, the configuration file is checked both syntactically and semantically (i.e. the data is correct). This tool uses the `libxml2` library to read, parse and validate the configuration file against the XML schema specification. Once validated by the library, the `xmcparser` performs a set of non-syntactical checks:

2235

- Memory area overlapping.
- Memory region overlapping.
- Memory area inside any region.
- Duplicated Partition's name and id.
- Allocated CPUs.
- Replicated port's names and id.
- Cyclic scheduling plan.
- Cyclic scheduling plan slot partition ids.
- Hardware IRQs allocated to partitions.
- IO port alignment.
- IO ports allocated to partitions.
- Allowed health monitoring actions.

2240

2245

9.1.1 xmcparser

2250

Compiles XtratuM XML configuration files

SYNOPSIS

xmcparser [-c] [-s xsd_file] [-o output_file] XM_CF.xml
xmcparser -d

DESCRIPTION

xmcparser reads an XtratuM XML configuration file and transforms it into a binary file which can be used directly by XtratuM at run time. **xmcparser** performs internally the following steps:

1. Parse the XML file.
2. Validate the XML data.
3. Generate a set of "C" data structures initialised with the XML data.
4. Compiles and links, using the target compiler, the "C" data structures. An ELF file is produced.
5. The data section which contains the data in binary format is extracted and copied to the output file.

OPTIONS

-d

Prints the default XML schema used to validate the XML configuration file.

-o file

Place output in file.

-s xsd_file

Use the XML schema xsd_file rather than the default XtratuM schema.

-c

Stop after the stage of "C" generation; do not compile. The output is in the form of a "C" file.

9.2 ELF to XEF (xmeformat)**9.2.1 xmeformat**

Creates and display information of XEF files

SYNOPSIS

xmeformat read [-h|-s|-m] file
xmeformat build [-m] [-o outfile] [-c] [-p payload_file] file

DESCRIPTION

xmeformat converts an ELF, or a binary file, into an XEF format (XtratuM Executable Format). An XEF file contains one or more segments. A segment is a block of data that shall be copied in a contiguous area of memory (when loaded in main memory). The content of the XEF can optionally be compressed.

An XEF file has a header and a set of segments. The segments corresponds to the allocatable sections of the source ELF file. In the header, there is a reserved area (16 bytes) to store user defined information. This information is called user payload.

build

A new XEF file is created, using file as input.

-m

The source file is not an ELF file but a user defined customisation. In this case, no consistency checks are performed.

Customisation files are used to attach data to the partitions (See the `xmpack` command). This data will be accessible to the partition at boot time. It is commonly used as partition defined run-time configuration parameters.

2290

-o file

Places output in file file.

-c

The XEF segments are compressed using the LSZZ algorithm.

2295

-p file

The first 16 bytes of the file are copied into the payload area of the XEF header. The size of the file shall be at least 16 bytes, otherwise an error is returned.

The MD5 sum value is printed if no errors.

read

2300

Shows the contents of the XEF file.

-h

Print the content of the header.

-s

Lists the segments and its attributes.

2305

-m

Lists the table of custom files. This options only works for partition and hypervisor XEF files.

USAGE EXAMPLES

Create a customisation file:

```
$ xmeformat build -m -o custom_file.xef data.in
b07715208bbfe72897a259619e7d7a6d custom_file.xef
```

2310

List the header of the XEF custom file:

```
$ xmeformat read -h custom_file.xef
XEF header:
signature: 0x24584546
version: 1.0.0
flags: XEF_DIGEST XEF_CONTENT_CUSTOMFILE
digest: b07715208bbfe72897a259619e7d7a6d
payload: 00 00 00 00 00 00 00 00
         00 00 00 00 00 00 00 00
file size: 232
segment table offset: 80
no. segments: 1
customFile table offset: 104
no. customFiles: 0
image offset: 104
image length: 127
XM image's header: 0x0
```

2315

2320

2325

Build the hypervisor XEF file:

```
$ xmeformat build -o xm_core.xef -c core/xm_core
```

2330

List the segments and headers of the XtratuM XEF file: `$ xmeformat read -s xm_core.xef` Segment table: 1 segments segment 0 physical address: 0x40000000 virtual address: 0x40000000 file size: 68520 compressed file size: 32923 (48.05%)

```

$ xmeformat read -h xm_core.xef
2335 XEF header:
      signature: 0x24584546
      version: 1.0.0
      flags: XEF_DIGEST XEF_COMPRESSED XEF_CONTENT_HYPERVISOR
      digest: 6698cfcf9311325e46e79ed50dfc9683
2340 payload: 00 00 00 00 00 00 00 00
           00 00 00 00 00 00 00 00
      file size: 33040
      segment table offset: 80
      no. segments: 1
2345 customFile table offset: 104
      no. customFiles: 1
      image offset: 112
      image length: 68520
      XM image's header: 0x40010b78
2350 compressed image length: 32928 (48.06%)

```

9.3 Container builder (xmpack)

9.3.1 xmpack

Create an XtratuM system image container

SYNOPSIS

xmpack build -h xm_file[@offset]:conf_file[@offset] [-p id:part_file[@offset][:custom_file[@offset]]*] + container

2355 **xmpack list** -c container

DESCRIPTION

2360 xmpack manipulates the XtratuM system container. The container is a simple filesystem designed to contain the XtratuM hypervisor core and zero or more XEF files. The container is an envelope to deploy all the system (hypervisor and partitions) from the host to the target. At boot time, the resident software is in charge of reading the contents of the container and coping the components to the RAM areas where the hypervisor and he partitions will be executed. Note that XtratuM has no knowledge about the container structure.

2365 The container is organised as a list of *components*. Each component is a list of XEF files. A component is used to store an executable unit, which can be: the XtratuM hypervisor or a partition. Each component is a list of one or more files. The first file shall be a valid XtratuM image (see the XtratuM binary file header) with the configuration file (once parsed and compiled into XEF format). The rest of the components are optional.

xmpack is a helper utility that can be used to deploy an XtratuM system. It is not mandatory to use this tool to deploy the application (hypervisor and the partitions) in the target machine.

The following checks are done:

- The binary image of the partitions fits into the allocated memory (as defined in the XM-CF).
- 2370 • The size of the customisation files fits into the area reserved by each partition.
- The memory allocated to XtratuM is big enough to hold the XtratuM image plus the configuration file.

build

A new *container* is created. Two kind of components can be defined:

-h to create an [H]ypervisor component:

The hypervisor entry is composed of the name of the XtratuM xef file and the binary configuration file (the result of processing the XM_Cf file). 2375

-p to create a [P]artition. The partition entries are composed of:

The *id* of the partition, as specified in the XM_Cf file. Note that this is the mechanism to bind the configuration description with the actual image of the partition. The *part_file* which shall contains the executable image. And zero or more *custom_files*. There shall be the same number of customisation files than that specified in the field *noCustomFiles* of the *xmImageHdr* structure. 2380

The elements that are part of each component are separated by ":".

By default, *xmpack* stores the files sequentially in the container. If the *offset* parameter is specified, then the file is placed at the given offset. The offset is defined with respect to the start of the container. The specified offset shall not overlap with existing data. The remaining files of the container will be placed after the end of this file. 2385

list

Shows the contents (components and the files of each component) of a container. If the option **-c** is given, the blocks allocated to each file are also shown.

USAGE EXAMPLES

2390

A new container with one hypervisor and one booting partition. The hypervisor container has two files: the hypervisor binary and the configuration table:

```
$ xmpack build build -h ../core/xm_core.bin:xm_ct.bin -p partition1.bin -o container
```

The same example but the second container has now two files: the partition image and a customisation file:

```
$ xmpack/xmpack build -h ../core/xm_core.bin:xm_cf.bin \
    -p partition1.bin:p1.cfg \
    -p partition2.bin:p2.cfg container.bin 2395
```

List the contents of the container:

```
$ xmpack list container.bin
<Package file="container.bin" version="1.0.0">
  <XMHypervisor file="../core/xm_core.bin" fileSize="97188" offset="0x0" size="97192" >
    <Module file="xm_cf.bin" size="8976" />
  </XMHypervisor>
  <Partition file="partition1.bin" fileSize="29996" offset="0x19eb8" size="30000" >
    <Module file="p1.cfg" size="16" />
  </Partition>
  <Partition file="partition2.bin" fileSize="30292" offset="0x213f8" size="30296" >
    <Module file="p2.cfg" size="16" />
  </Partition>
</Package> 2400
2405
2410
```

9.4 Bootable image creator (rswbuild)

9.4.1 rswbuild

Create a bootable image

SYNOPSIS

rswbuild container bootable

2415 DESCRIPTION

rswbuild is a shell script that creates a bootable file by combining the resident software code with the container file. The container shall be a valid file created with the `xmpack` tool.

The resident software object file is read from the distribution directory pointer by the `$XTRATUM_PATH` variable.

USAGE EXAMPLES

2420 `rswbuild container resident_sw`

Chapter 10

Security issues

This chapter introduces several security issues related with XtratuM which should be taken into account by partition developers.

10.1 Invoking a hypercall from libXM

Invoking a hypercall requires a non-standard protocol which must be directly implemented in assembly code.

LibXM is a partition-level “C” library deployed jointly with XtratuM aiming to hide this complexity and ease the development of “C” partitions. 2425

From the security point of view, XtratuM implements two stacks for each partition: one managed by the partition (user context) and another, internal, managed directly by XtratuM (supervisor context). The partition stack is used by the libXM to prepare the call to XtratuM (pretty much like the gLibc does). Once the hypercall service is invoked, XtratuM changes the stack to its own stack. This second stack may contain sensitive information, but it is located inside the memory space of XtratuM (not exposed). It is normal to observe that the partition stack is modified when a hypercall is called, however this behaviour is far from being considered an actual security issue. 2430

10.2 Preventing covert/side channels due to scheduling slot overrun

This version of XtratuM is non-preemptible: once the kernel starts an activity (e.g. a service), it cannot be interrupted until its completion. This behaviour includes any hypercall invocation: if a partition calls an hypercall just before a partition context switch must be performed, XtratuM will not carry out the action until the hypercall is finished. This overrun can be exploited to gain information. The information is obtained by measuring the temporal cost of the last hypercall. There are two types of information that can be retrieved: 2435

1. Whether the target partition was executing an hypercall at the end of the slot or not. If the spy partition start at the nominal slot start time or not. 2440
2. In the case of being executing a hypercall, how much time XtratuM needed to attend it: the cost of the last hypercall.

In the case of a covert channel, the maximum bandwidth is determined by the duration of the longest hypercall divided by the clock resolution. A rough estimation (supposing that the maximum message length is 4096 Bytes) is 4 bits at each partition context switch. 2445

In the case of a side channel, the bandwidth is drastically reduced due to the uncertainty/randomness introduced by the execution of the target partition.

There are several strategies to address this issue:

1. At integrator level: design a scheduling plan that lets some idle time between the end of a slot and the beginning of the next one. The Xoncrete scheduling tool is able to implement that solution automatically. Figure 10.1 shows two scenarios: scenario 1 where the integrator has left no spare time between one partition slot and the next one, enabling the partition in light grey to overrun the start of the dark gray one. Scenario 2 sketches the same case but leaving spare time between one slot and the next one. So, in this case, execution overruns can not occur.
2. At partition level: stop invoking hypercalls some time before the end of the slot. This way, there will be no hypercalls being executed when the slot finishes, so the next partition will always start with no delay.
3. At hypervisor level:
 - (a) Change the design of XtratuM to make it preemptable. Hypercalls would be interrupted when the end of the slot is reached, and later resumed when the partition is active again.
 - (b) Implement the partial preemptability in XtratuM (voluntary preemption). XtratuM is by default atomic (non-preemptable) but at some designated safe places in the code, the preemption is allowed.

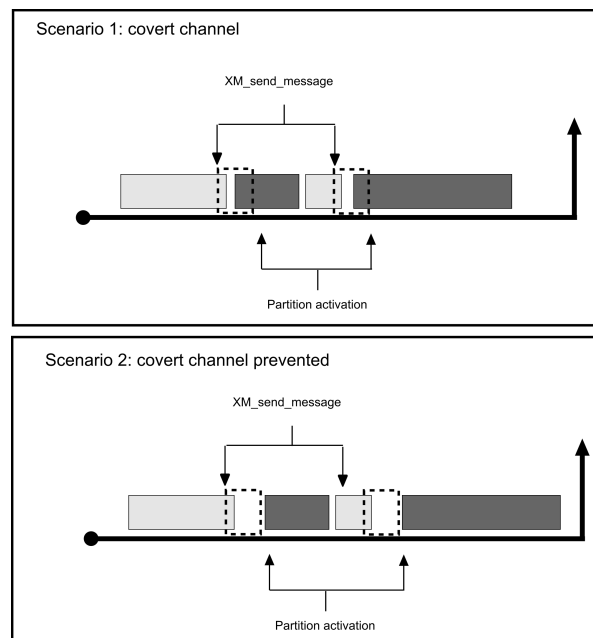


Figure 10.1: Covert channel caused by an incorrect scheduling plan and a solution.

Appendix A

XML Schema Definition

A.1 XML Schema file

basicstyle

```
1 <?xml version="1.0"?>
2 <xs:schema targetNamespace="http://www.xratum.org/xm-arm-2.x"
3           xmlns:xs="http://www.w3.org/2001/XMLSchema"
4           xmlns="http://www.xratum.org/xm-arm-2.x"
5           elementFormDefault="qualified"
6           attributeFormDefault="unqualified">
7   <!-- Basic types definition -->
8   <xs:simpleType name="id_t">
9     <xs:restriction base="xs:integer">
10       <xs:minInclusive value="0"/>
11     </xs:restriction>
12   </xs:simpleType>
13   <xs:simpleType name="idString_t">
14     <xs:restriction base="xs:string">
15       <xs:minLength value="1"/>
16     </xs:restriction>
17   </xs:simpleType>
18   <xs:simpleType name="hwIrqId_t">
19     <xs:restriction base="xs:integer">
20       <xs:minInclusive value="0"/>
21       <xs:maxExclusive value="
22         96
23       "/>
24     </xs:restriction>
25   </xs:simpleType>
26   <xs:simpleType name="hwIrqIdList_t">
27     <xs:list itemType="hwIrqId_t"/>
28   </xs:simpleType>
29   <xs:simpleType name="idList_t">
30     <xs:list itemType="id_t"/>
31   </xs:simpleType>
32   <xs:simpleType name="hex_t">
33     <xs:restriction base="xs:string">
34       <xs:pattern value="0x[0-9a-fA-F]+" />
35     </xs:restriction>
36   </xs:simpleType>
37   <xs:simpleType name="version_t">
```

```

38     <xs:restriction base="xs:string">
39         <xs:pattern value="[0-9]+\.[0-9]+\.[0-9]+" />
40     </xs:restriction>
41 </xs:simpleType>
42 <xs:simpleType name="freqUnit_t">
43     <xs:restriction base="xs:string">
44         <xs:pattern value="[0-9]+(\.[0-9]+)?([MK][Hh]z)" />
45     </xs:restriction>
46 </xs:simpleType>
47 <xs:simpleType name="processorFeatures_t">
48     <xs:restriction base="xs:string">
49         <xs:enumeration value="XM_CPU_LEON2_WA1" />
50         <xs:enumeration value="none" />
51     </xs:restriction>
52 </xs:simpleType>
53 <xs:simpleType name="discipline_t">
54     <xs:restriction base="xs:string">
55         <xs:enumeration value="FIFO" />
56         <xs:enumeration value="PRIORITY" />
57     </xs:restriction>
58 </xs:simpleType>
59 <xs:simpleType name="processorFeaturesList_t">
60     <xs:list itemType="processorFeatures_t" />
61 </xs:simpleType>
62 <xs:simpleType name="partitionFlags_t">
63     <xs:restriction base="xs:string">
64         <xs:enumeration value="system" />
65         <xs:enumeration value="fp" />
66         <xs:enumeration value="boot" />
67         <xs:enumeration value="icache_disabled" />
68         <xs:enumeration value="dcache_disabled" />
69         <xs:enumeration value="none" />
70     </xs:restriction>
71 </xs:simpleType>
72 <xs:simpleType name="partitionFlagsList_t">
73     <xs:list itemType="partitionFlags_t" />
74 </xs:simpleType>
75 <xs:simpleType name="sizeUnit_t">
76     <xs:restriction base="xs:string">
77         <xs:pattern value="[0-9]+(\.[0-9]+)?([MK]?B)" />
78     </xs:restriction>
79 </xs:simpleType>
80 <xs:simpleType name="timeUnit_t">
81     <xs:restriction base="xs:string">
82         <xs:pattern value="[0-9]+(\.[0-9]+)?([mu]?[sS])" />
83     </xs:restriction>
84 </xs:simpleType>
85
86 <xs:simpleType name="hmString_t">
87     <xs:restriction base="xs:string">
88         <xs:enumeration value="XM_HM_EV_INTERNAL_ERROR" />
89         <xs:enumeration value="XM_HM_EV_UNEXPECTED_TRAP" />
90         <xs:enumeration value="XM_HM_EV_PARTITION_ERROR" />
91         <xs:enumeration value="XM_HM_EV_PARTITION_INTEGRITY" />
92         <xs:enumeration value="XM_HM_EV_MEM_PROTECTION" />
93         <xs:enumeration value="XM_HM_EV_OVERRUN" />
94         <xs:enumeration value="XM_HM_EV_SCHED_ERROR" />
95         <xs:enumeration value="XM_HM_EV_WATCHDOG_TIMER" />
96         <xs:enumeration value="XM_HM_EV_INCOMPATIBLE_INTERFACE" />

```

```

97     <xs:enumeration value="XM_HM_EV_ARM_UNDEF_INSTR"/>
98     <xs:enumeration value="XM_HM_EV_ARM_PREFETCH_ABORT"/>
99     <xs:enumeration value="XM_HM_EV_ARM_DATA_ABORT"/>
100    <xs:enumeration value="XM_HM_EV_ARM_DATA_ALIGNMENT_FAULT"/>
101    <xs:enumeration value="XM_HM_EV_ARM_DATA_BACKGROUND_FAULT"/>
102    <xs:enumeration value="XM_HM_EV_ARM_DATA_PERMISSION_FAULT"/>
103    <xs:enumeration value="XM_HM_EV_ARM_INSTR_ALIGNMENT_FAULT"/>
104    <xs:enumeration value="XM_HM_EV_ARM_INSTR_BACKGROUND_FAULT"/>
105    <xs:enumeration value="XM_HM_EV_ARM_INSTR_PERMISSION_FAULT"/>
106  </xs:restriction>
107 </xs:simpleType>
108
109
110 <xs:simpleType name="hmAction_t">
111   <xs:restriction base="xs:string">
112     <xs:enumeration value="XM_HM_AC_IGNORE"/>
113     <xs:enumeration value="XM_HM_AC_SHUTDOWN"/>
114     <xs:enumeration value="XM_HM_AC_PARTITION_COLD_RESET"/>
115     <xs:enumeration value="XM_HM_AC_PARTITION_WARM_RESET"/>
116     <xs:enumeration value="XM_HM_AC_HYPERVISOR_COLD_RESET"/>
117     <xs:enumeration value="XM_HM_AC_HYPERVISOR_WARM_RESET"/>
118     <xs:enumeration value="XM_HM_AC_SUSPEND"/>
119     <xs:enumeration value="XM_HM_AC_HALT"/>
120     <xs:enumeration value="XM_HM_AC_PROPAGATE"/>
121     <xs:enumeration value="XM_HM_AC_SWITCH_TO_MAINTENANCE" />
122   </xs:restriction>
123 </xs:simpleType>
124
125 <xs:simpleType name="memAreaFlags_t">
126   <xs:restriction base="xs:string">
127     <xs:enumeration value="unmapped"/>
128     <xs:enumeration value="read-only"/>
129     <xs:enumeration value="uncacheable"/>
130     <xs:enumeration value="rom"/>
131     <xs:enumeration value="flag0"/>
132     <xs:enumeration value="flag1"/>
133     <xs:enumeration value="flag2"/>
134     <xs:enumeration value="flag3"/>
135     <xs:enumeration value="none"/>
136   </xs:restriction>
137 </xs:simpleType>
138 <xs:simpleType name="memAreaFlagsList_t">
139   <xs:list itemType="memAreaFlags_t"/>
140 </xs:simpleType>
141 <xs:simpleType name="slotFlags_t">
142   <xs:restriction base="xs:string">
143     <xs:enumeration value="periodStart"/>
144   </xs:restriction>
145 </xs:simpleType>
146 <xs:simpleType name="slotFlagsList_t">
147   <xs:list itemType="slotFlags_t"/>
148 </xs:simpleType>
149 <xs:simpleType name="memRegion_t">
150   <xs:restriction base="xs:string">
151     <xs:enumeration value="sdram"/>
152     <xs:enumeration value="stram"/>
153     <xs:enumeration value="rom"/>
154   </xs:restriction>
155 </xs:simpleType>

```

```

156 <xs:simpleType name="portType_t">
157   <xs:restriction base="xs:string">
158     <xs:enumeration value="queuing"/>
159     <xs:enumeration value="sampling"/>
160   </xs:restriction>
161 </xs:simpleType>
162 <xs:simpleType name="direction_t">
163   <xs:restriction base="xs:string">
164     <xs:enumeration value="source"/>
165     <xs:enumeration value="destination"/>
166   </xs:restriction>
167 </xs:simpleType>
168 <xs:simpleType name="yntf_t">
169   <xs:restriction base="xs:string">
170     <xs:enumeration value="yes"/>
171     <xs:enumeration value="no"/>
172     <xs:enumeration value="true"/>
173     <xs:enumeration value="false"/>
174   </xs:restriction>
175 </xs:simpleType>
176 <!-- End Types -->
177 <!-- Elements -->
178 <!-- Hypervisor -->
179 <xs:complexType name="hypervisor_e">
180   <xs:all>
181     <xs:element name="PhysicalMemoryArea" type="hypMemoryArea_e"/>
182     <xs:element name="HealthMonitor" type="healthMonitor_e" minOccurs="0" />
183   </xs:all>
184   <xs:attribute name="console" type="idString_t" use="optional" />
185   <xs:attribute name="features" type="xs:string" use="optional" />
186 </xs:complexType>
187 <!-- Rsw -->
188 <xs:complexType name="rsw_e">
189   <xs:all>
190     <xs:element name="PhysicalMemoryAreas" type="memoryArea_e"/>
191   </xs:all>
192 </xs:complexType>
193 <!-- Partition -->
194 <xs:complexType name="partition_e">
195   <xs:all>
196     <xs:element name="PhysicalMemoryAreas" type="memoryArea_e"/>
197     <xs:element name="TemporalRequirements" minOccurs="0">
198       <xs:complexType>
199         <xs:attribute name="period" type="timeUnit_t" use="required"/>
200         <xs:attribute name="duration" type="timeUnit_t" use="required"/>
201       </xs:complexType>
202     </xs:element>
203     <xs:element name="HealthMonitor" type="healthMonitor_e" minOccurs="0" />
204     <xs:element name="HwResources" type="hwResources_e" minOccurs="0" />
205     <xs:element name="PortTable" type="partitionPorts_e" minOccurs="0" />
206   </xs:all>
207   <xs:attribute name="id" type="id_t" use="required"/>
208   <xs:attribute name="name" type="idString_t" use="optional" />
209   <xs:attribute name="console" type="idString_t" use="optional" />
210   <xs:attribute name="flags" type="partitionFlagsList_t" use="optional" default="none"
211     />
212 </xs:complexType>
213 <!-- Communication Ports -->
214 <xs:complexType name="partitionPorts_e">

```

```

214     <xs:sequence minOccurs="0" maxOccurs="unbounded">
215         <xs:element name="Port">
216             <xs:complexType>
217                 <xs:attribute name="name" type="idString_t" use="required"/>
218                 <xs:attribute name="direction" type="direction_t" use="required"/>
219                 <xs:attribute name="type" type="portType_t" use="required"/>
220                 <xs:attribute name="discipline" type="discipline_t" use="optional" />
221             </xs:complexType>
222         </xs:element>
223     </xs:sequence>
224 </xs:complexType>
225 <!-- Channels -->
226 <xs:complexType name="channels_e">
227     <xs:sequence minOccurs="0" maxOccurs="unbounded">
228         <xs:choice>
229             <xs:element name="Ipvi">
230                 <xs:complexType>
231                     <xs:attribute name="id" type="id_t" use="required"/>
232                     <xs:attribute name="sourceId" type="id_t" use="required"/>
233                     <xs:attribute name="destinationId" type="idList_t" use="required"/>
234                 </xs:complexType>
235             </xs:element>
236             <xs:element name="SamplingChannel">
237                 <xs:complexType>
238                     <xs:sequence minOccurs="1">
239                         <xs:choice>
240                             <xs:element name="Source" type="ipcPort_e" />
241                         </xs:choice>
242                     <xs:sequence minOccurs="1" maxOccurs="unbounded">
243                         <xs:choice>
244                             <xs:element name="Destination" type="ipcPort_e"/>
245                         </xs:choice>
246                     </xs:sequence>
247                 </xs:sequence>
248                 <xs:attribute name="maxMessageLength" type="sizeUnit_t" use="required"/>
249                 <xs:attribute name="refreshPeriod" type="timeUnit_t" use="optional" default="
250                     0s"/>
251             </xs:complexType>
252             </xs:element>
253             <xs:element name="QueuingChannel">
254                 <xs:complexType>
255                     <xs:sequence minOccurs="1">
256                         <xs:choice>
257                             <xs:element name="Source" type="ipcPort_e" />
258                         </xs:choice>
259                     <xs:choice>
260                         <xs:element name="Destination" type="ipcPort_e" />
261                     </xs:choice>
262                 </xs:sequence>
263                 <xs:attribute name="maxMessageLength" type="sizeUnit_t" use="required"/>
264                 <xs:attribute name="maxNoMessages" type="xs:positiveInteger" use="required"/>
265                 <xs:attribute name="maxTimeExpiration" type="timeUnit_t" use="optional"
266                     default="0s"/>
267             </xs:complexType>
268             </xs:element>
269         </xs:choice>
270     </xs:sequence>
271 </xs:complexType>
272 <!-- Devices -->

```

```

271 <xs:complexType name="devices_e">
272   <xs:sequence minOccurs="0" maxOccurs="unbounded">
273     <xs:choice>
274       <xs:element name="MemoryBlock" minOccurs="0">
275         <xs:complexType>
276           <xs:attribute name="name" type="idString_t" use="required"/>
277           <xs:attribute name="start" type="hex_t" use="required"/>
278           <xs:attribute name="size" type="sizeUnit_t" use="required"/>
279         </xs:complexType>
280       </xs:element>
281       <xs:element name="Uart" minOccurs="0">
282         <xs:complexType>
283           <xs:attribute name="name" type="idString_t" use="required"/>
284           <xs:attribute name="id" type="idString_t" use="required"/>
285           <xs:attribute name="baudRate" type="xs:positiveInteger" use="required"/>
286         </xs:complexType>
287       </xs:element>
288       <xs:element name="Null" minOccurs="0">
289         <xs:complexType>
290           <xs:attribute name="name" type="idString_t" use="optional" />
291         </xs:complexType>
292       </xs:element>
293     </xs:choice>
294   </xs:sequence>
295 </xs:complexType>
296 <!-- IPC Port -->
297 <xs:complexType name="ipcPort_e">
298   <xs:attribute name="partitionId" type="id_t" use="required"/>
299   <xs:attribute name="partitionName" type="idString_t" use="optional" />
300   <xs:attribute name="portName" type="idString_t" use="required"/>
301 </xs:complexType>
302 <!-- Hw Description -->
303 <xs:complexType name="hwDescription_e">
304   <xs:sequence>
305     <xs:element name="MemoryLayout" type="memoryLayout_e"/>
306     <xs:element name="ProcessorTable">
307       <xs:complexType>
308         <xs:sequence minOccurs="1" maxOccurs="256">
309           <xs:element name="Processor" type="processor_e" />
310         </xs:sequence>
311       </xs:complexType>
312     </xs:element>
313     <xs:element name="Devices" type="devices_e"/>
314   </xs:sequence>
315 </xs:complexType>
316 <!-- Processor -->
317 <xs:complexType name="processor_e">
318   <xs:all>
319     <xs:element name="CyclicPlanTable" type="cyclicPlan_e"/>
320   </xs:all>
321   <xs:attribute name="id" type="id_t" use="required"/>
322   <xs:attribute name="frequency" type="freqUnit_t" use="optional" />
323   <xs:attribute name="features" type="processorFeaturesList_t" use="optional" default=
    "none"/>
324   <xs:attribute name="console" type="xs:string" use="optional" />
325 </xs:complexType>
326 <!-- HwResource -->
327 <xs:complexType name="hwResources_e">
328   <xs:all>

```



```

329     <xs:element name="IoPorts" type="ioPorts_e" minOccurs="0" />
330     <xs:element name="Interrupts" minOccurs="0">
331         <xs:complexType>
332             <xs:attribute name="lines" type="hwIrqIdList_t" use="required"/>
333         </xs:complexType>
334     </xs:element>
335     <xs:element name="APBDev" type="APBDeviceType" minOccurs="0"/>
336 </xs:all>
337 </xs:complexType>
338 <!-- Io Ports -->
339 <xs:complexType name="ioPorts_e">
340     <xs:sequence minOccurs="0" maxOccurs="unbounded">
341         <xs:choice>
342             <xs:element name="Range">
343                 <xs:complexType>
344                     <xs:attribute name="base" type="hex_t" use="required"/>
345                     <xs:attribute name="noPorts" type="xs:positiveInteger" use="required"/>
346                 </xs:complexType>
347             </xs:element>
348             <xs:element name="Restricted">
349                 <xs:complexType>
350                     <xs:attribute name="address" type="hex_t" use="required"/>
351                     <xs:attribute name="mask" type="hex_t" use="optional" default="0x0"/>
352                 </xs:complexType>
353             </xs:element>
354         </xs:choice>
355     </xs:sequence>
356 </xs:complexType>
357 <!-- APBId -->
358 <xs:simpleType name="APBDevice_e">
359     <xs:restriction base="xs:string"/>
360 </xs:simpleType>
361 <xs:complexType name="APBDeviceType">
362     <xs:attribute name="device" type="APBDevice_e"/>
363 </xs:complexType>
364 <!-- CyclicPlan -->
365 <xs:complexType name="cyclicPlan_e">
366     <xs:sequence minOccurs="1" maxOccurs="unbounded">
367         <xs:element name="Plan" type="plan_e" />
368     </xs:sequence>
369 </xs:complexType>
370 <!-- Plan -->
371 <xs:complexType name="plan_e">
372     <xs:sequence minOccurs="1" maxOccurs="unbounded">
373         <xs:element name="Slot">
374             <xs:complexType>
375                 <xs:attribute name="id" type="id_t" use="required"/>
376                 <xs:attribute name="start" type="timeUnit_t" use="required"/>
377                 <xs:attribute name="duration" type="timeUnit_t" use="required"/>
378                 <xs:attribute name="partitionId" type="id_t" use="required"/>
379                 <xs:attribute name="vCpuId" type="id_t" use="optional" default="0"/>
380                 <xs:attribute name="flags" type="slotFlagsList_t" use="optional"/>
381             </xs:complexType>
382         </xs:element>
383     </xs:sequence>
384     <xs:attribute name="name" type="idString_t" use="optional"/>
385     <xs:attribute name="id" type="id_t" use="required"/>
386     <xs:attribute name="majorFrame" type="timeUnit_t" use="required"/>
387 </xs:complexType>

```

```

388 <!-- Health Monitor -->
389 <xs:complexType name="healthMonitor_e">
390   <xs:sequence minOccurs="1" maxOccurs="unbounded">
391     <xs:element name="Event">
392       <xs:complexType>
393         <xs:attribute name="name" type="hmString_t" use="required"/>
394         <xs:attribute name="action" type="hmAction_t" use="required"/>
395         <xs:attribute name="log" type="yntf_t" use="required"/>
396       </xs:complexType>
397     </xs:element>
398   </xs:sequence>
399 </xs:complexType>
400 <!-- Memory Layout -->
401 <xs:complexType name="memoryLayout_e">
402   <xs:sequence minOccurs="1" maxOccurs="unbounded">
403     <xs:element name="Region">
404       <xs:complexType>
405         <xs:attribute name="type" type="memRegion_t" use="required"/>
406         <xs:attribute name="start" type="hex_t" use="required"/>
407         <xs:attribute name="size" type="sizeUnit_t" use="required"/>
408       </xs:complexType>
409     </xs:element>
410   </xs:sequence>
411 </xs:complexType>
412 <!-- Hypervisor Memory Area -->
413 <xs:complexType name="hypMemoryArea_e">
414   <xs:attribute name="size" type="sizeUnit_t" use="required"/>
415   <xs:attribute name="flags" type="memAreaFlagsList_t" use="optional"/>
416 </xs:complexType>
417 <!-- Memory Area -->
418 <xs:complexType name="memoryArea_e">
419   <xs:sequence minOccurs="1" maxOccurs="unbounded">
420     <xs:element name="Area">
421       <xs:complexType>
422         <xs:attribute name="name" type="idString_t" use="optional" />
423         <xs:attribute name="start" type="hex_t" use="required"/>
424         <xs:attribute name="size" type="sizeUnit_t" use="required"/>
425         <xs:attribute name="flags" type="memAreaFlagsList_t" use="optional" default="
         none"/>
426         <xs:attribute name="mappedAt" type="hex_t" use="optional"/> <!-- default="" -->
427       </xs:complexType>
428     </xs:element>
429   </xs:sequence>
430 </xs:complexType>
431 <!-- Root Element -->
432 <xs:element name="SystemDescription">
433   <xs:complexType>
434     <xs:all>
435       <xs:element name="HwDescription" type="hwDescription_e" />
436       <xs:element name="XMHypervisor" type="hypervisor_e"/>
437       <xs:element name="ResidentSw" type="rsw_e" minOccurs="0"/>
438       <xs:element name="PartitionTable">
439         <xs:complexType>
440           <xs:sequence maxOccurs="unbounded">
441             <xs:element name="Partition" type="partition_e" />
442           </xs:sequence>
443         </xs:complexType>
444       </xs:element>
445       <xs:element name="Channels" type="channels_e" minOccurs="0" />

```

```

446     </xs:all>
447     <xs:attribute name="version" type="version_t" use="required"/>
448     <xs:attribute name="name" type="idString_t" use="required"/>
449   </xs:complexType>
450 </xs:element>
451 <!-- End Root Element -->
452 <!-- Elements -->
453 </xs:schema>

```

Listing A.1: xmc.xsd

A.2 Configuration file example

```

<SystemDescription xmlns="http://www.xtratum.org/xm-arm-2.x" version="1.0.0" name=
  "example">
  <HwDescription>
    <MemoryLayout>
      <Region type="rom" start="0x0" size="1MB" />
      <Region type="sdram" start="0x00100000" size="1023MB" />
    </MemoryLayout>
    <ProcessorTable>
      <Processor id="0" frequency="400Mhz">
        <CyclicPlanTable>
          <Plan id="0" majorFrame="500ms">
            <Slot id="0" start="0ms" duration="250ms" partitionId="0"/>
          </Plan>
        </CyclicPlanTable>
      </Processor>
      <Processor id="1" frequency="400Mhz">
        <CyclicPlanTable>
          <Plan id="0" majorFrame="500ms">
            <Slot id="0" start="0ms" duration="250ms" partitionId="1"/>
          </Plan>
        </CyclicPlanTable>
      </Processor>
    </ProcessorTable>
    <Devices>
      <Uart id="1" baudRate="115200" name="Uart" />
    </Devices>
  </HwDescription>
  <XMHypervisor console="Uart">
    <PhysicalMemoryArea size="512KB" />
  </XMHypervisor>
  <PartitionTable>
    <Partition id="0" name="Partition0" flags="boot" console="Uart">
      <PhysicalMemoryAreas>
        <Area start="0x10000000" size="256KB" />
      </PhysicalMemoryAreas>
    </Partition>
    <Partition id="1" name="Partition1" flags="boot" console="Uart">
      <PhysicalMemoryAreas>
        <Area start="0x14000000" size="256KB" />
      </PhysicalMemoryAreas>
    </Partition>
  </PartitionTable>
</SystemDescription>

```



Listing A.2: XML configuration file example

GNU Free Documentation License

Version 1.2, November 2002

Copyright © 2000,2001,2002 Free Software Foundation, Inc.

2510

51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document “free” in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

2515

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

2520

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

2525

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The “**Document**”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “**you**”. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

2530

A “**Modified Version**” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “**Secondary Section**” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

2535

2540

The “**Invariant Sections**” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the

above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The “**Cover Texts**” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A “**Transparent**” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not “Transparent” is called “**Opaque**”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The “**Title Page**” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

A section “**Entitled XYZ**” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “**Acknowledgements**”, “**Dedications**”, “**Endorsements**”, or “**History**”.) To “**Preserve the Title**” of such a section when you modify the Document means that it remains a section “Entitled XYZ” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

2690

If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

2695

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

2700

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

2705

ADDENDUM: How to use this License for your documents

2710

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (c)  YEAR  YOUR NAME.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.2
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
A copy of the license is included in the section entitled ‘‘GNU
Free Documentation License’’.
```

2715

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the “with ... Texts.” line with this:

2720

```
with the Invariant Sections being LIST THEIR TITLES, with the
Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

2725

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

This page is intentionally left blank.

Applicable and reference documents

2730

Reference documents

- [1] Fent Innovative Software Solution. XM-ARM: Software Starter Guide. Tech. rep. 14-035-03.009.sum.02. Fent Innovative Software Solutions, S.L., January, 2016 (cit. on p. 43).

This page is intentionally left blank.

Glossary of Terms and Acronyms

2735

Glossary

Abbreviated terms

This page is intentionally left blank.