# XtratuM Hypervisor for SPARCv8

## User Manual

Miguel Masmano, Alfons Crespo, Javier Coronel

July, 2017
Reference: 14-036.008.sum.um

fentISS
FENT Innovative Software Solutions

This page is intentionally left blank.

# DOCUMENT CONTROL PAGE

**TITLE:** XtratuM Hypervisor for SPARCv8: User Manual

**AUTHOR/S:** Miguel Masmano, Alfons Crespo, Javier Coronel

**LAST PAGE NUMBER:** 146

**VERSION OF SOURCE CODE:** XtratuM for SPARCv8()

**REFERENCE ID:** 14-036.008.sum.um

**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. Contributions to this documentation (new material, suggestions or corrections) are welcome.

**REFERENCING THIS DOCUMENT:**

```
@techreport {14-036.008.sum.um,
       title = {XtratuM Hypervisor for SPARCv8:  User Manual},
       author = { Miguel Masmano and Alfons Crespo and Javier Coronel},
       institution = {Fent Innovative Software Solutions, S.L.},
       number = {14-036.008.sum.um},
       year={July, 2017},
}
```



**Changes:**

| Version | Date | Comments |
|---|---|---|
| 0.1 | February, 2010 | [xm-3-usermanual-022] Initial document, based on the document XM-usermanual-022. Code version 3.1.0.<br>• Major changes in the ABI chapter (Binary interfaces).<br>• Major changes in the tools chapter.<br>• Internal object programming reworked. No impact on the libxm API. |
| 0.2 | May, 2010 | [xm-3-usermanual-022b]. Code version 3.1.2<br>• Multi-plan support. Sections 2.5.1 and 5.7.<br>• The PIT (Partition Information Table) has been removed. The PIT fields have been moved to the PCT. The PCT is not read-only.<br>• Message notification support has been removed. |

| Version | Date | Comments |
|---|---|---|
| 0.3 | October, 2010 | [xm-3-usermanual-022c]. Code version 3.1.2<br>• Global line numbering feature has been added. |
| 0.4 | February, 2011 | [xm-3-usermanual-022e]. Code version 3.2.0<br>• Several typos corrected. |
| 1.0 | September, 2011 | [xm-3-usermanual-022f]. Code version 3.3.0<br>• Final version |
| 1.1 | December, 2011 | [xm-3-usermanual-022g]. Code version 3.3.2<br>• Updated version with new section on cache management. |
| 1.2 | March, 2012 | [xm-3-usermanual-022h]. Code version 3.3.2b<br>• LEON2 with MMU support.<br>• Memory management updated: MMU page management and mappedAt attribute.<br>• XtratuM image header removes page table fields that are not required. |
| 1.3 | July, 2012 | [xm-3-usermanual-63]. Code version 3.9.0<br>• Updated tracing support section.<br>• Updated HM support section.<br>• Updated XM configuration section.<br>• Updated XML Schema description.<br>• Added Internal instrumentation section. |
| 1.4 | September, 2012 | [xm-3-usermanual-63c]. Code version 3.9.1<br>• Updated HM support section. |
| 1.5 | February, 2013 | [xm-3-usermanual-63d]. Code version 3.9.4<br>• AMP architecture subsection added.<br>• XML updated to reflect external sampling and queuing ports.<br>• ampbuilder.py tool added. |
| 1.6 | September, 2013 | [fnts-xm-um-2a] reference changed using FentISS convention. |
| 1.7 | January, 2014 | [fnts-xm-um-2c]. Code version 3.9.7<br>• SMP architecture subsection updated to add attribute console.<br>• Menuconfig options updated with the new ones.<br>• XML configuration items updated. |
| 1.8 | January, 2015 | [14-036.008.sum]. Code version 1.0.1.<br>• XML schema updated. entryPoint attribute removed. |
| 1.9 | January, 2015 | [14-036.008.sum]. Code version 1.0.2.<br>• Partition reset subsection updated with partition's default entry point retrieval process. |

| Version | Date | Comments |
|---|---|---|
| 2.0 | September, 2015 | [14-036.008.sum]. Code version 1.0.5.<br>• [SPR-080915-02] added 2.5.2, updated 8.3.4.<br>• [SPR-080915-01] Shared value for "@flags" attribute removed from 2.6, 5.16, 8.3.6.<br>• [SPR-010915-01] MIL-STD-1553 driver description added in 2.18.1 and 8.1 updated.<br>• [SPR-280715-01] "@entryPoint" added to 8.3.6. |
| 2.1 | November, 2015 | [14-036.008.sum]. Code version 1.0.6.<br>• [SPR-221015-01] 5.16.1 added.<br>• [SPR-221115-01] 2.5.2 updated.<br>• [SPR-061115-01] 7.1 updated.<br>• [SPR-241115-02] 2.1.1 added.<br>• [SPR-031215-01] 5.5 updated.<br>• [SPR-111215-03] 2.9.1 updated. |
| 2.2 | April, 2016 | [14-036.008.sum]. Code version 1.0.7.<br>• [SPR-180216-03] 10.1.1 updated.<br>• [CP-250216-01] 5.13 added.<br>• [CP-160411-01] 5.5 updated.<br>• [CP-080316-01] 2.18.1, 8.1 updated. |
| 2.3 | July, 2017 | [14-036.008.sum]. Code version 1.0.8.<br>• [SPR-160408-01] 2.1 update "System's states and transitions" figure.<br>• [SPR-160613-01] 5.17 updated.<br>• [SPR-160720-01] Update the hardware state assumed by XtratuM when the bootloader jumps to the XtratuM entrypoint in 7.1.<br>• [SPR-121115-02] 2.14 and 2.15 updated. |

This page is intentionally left blank.

# Contents

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 LEON3 (SPARC v8) 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:

```
static inline void XM_sparcv8_set_psr(xm_u32_t flags) {
   __asm__ __volatile__("mov "TO_STR(sparcv8_set_psr_nr)", %%o0\n\t" \
           "mov %0, %%o1\n\t" \
           __DO_XMAHC :: "r"(flags) : "o0", "o1");
}
```

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 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 the research and the market is 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.

During the porting of XtratuM to the LEON2 and LEON3 processors, we have also adapted the XtratuM API and internal operations to resemble ARINC-653 standard. It is not our intention to convert XtratuM in 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.

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 formats of the different components involved in the system development: system image, partition format, partition tables. The chapter ends with the 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 of the preliminar tools developed to analyse system configuration schemas (XML format) and generate the appropriate internal structures to configure XtratuM for a specific payload.

## 1.1 History

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 for the rest of the system.

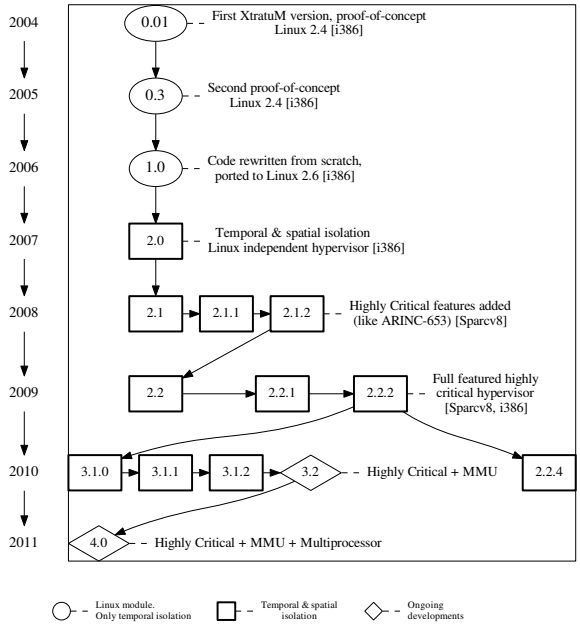



Figure 1.1: XtratuM evolution.

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 constrains 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.
- There is not a strong memory isolation between partitions.
- Linux is executed in processor supervisor mode.
- Linux is responsible of 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 never released.

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.
- 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.
- 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.

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 subsistem.
- Virtual interrupt subsystem has been rewritten from scratch.
- 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.

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

The current development version is 3.3. Thi version is stil under active development.

In what follows, the name XtratuM will be used to refer to the version 3 and more specifically to 3.2 of XtratuM.

This page is intentionally left blank.

# 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.

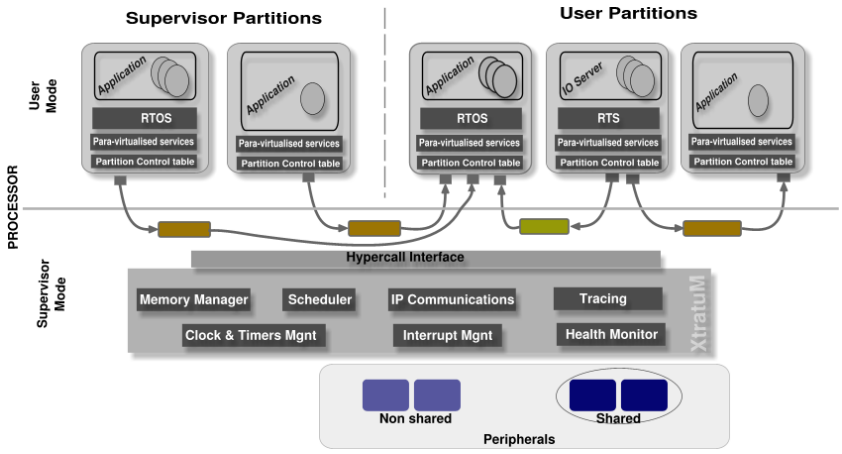



Figure 2.1: XtratuM architecture.

The main components of this architecture are:

- Hypervisor: XtratuM provides virtualisation services to partitions. It is executed in processor supervisor mode 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 are handled by XtratuM and, depending on the interrupt nature, propagated to the partitions. XtratuM provides a 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. 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 *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 so the the provided and non-provided services of XtratuM must be taken into account.

**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).

## 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, to 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).

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).

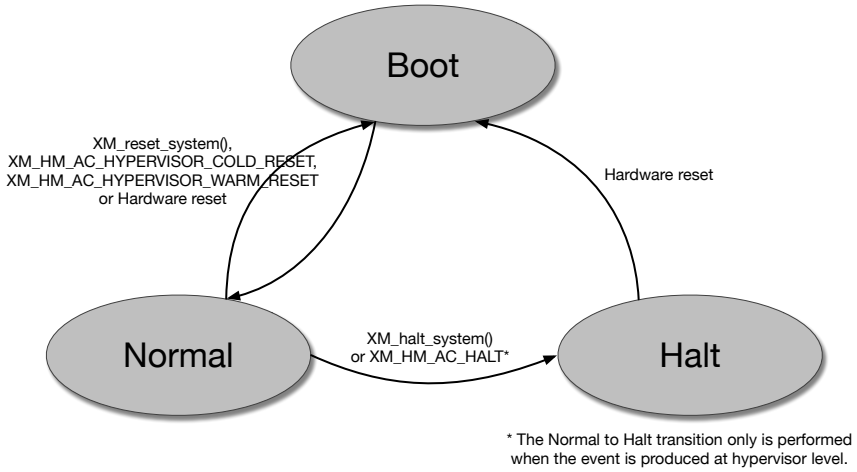



Figure 2.2: System's states and transitions.

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.

### 2.1.1 Cold system reset behaviour

XtratuM permits to configure during its compilation the behaviour of the cold system reset service. When the option *Hypervisor/Jump to user function on cold reset system option* is not selected, on a successful cold system reset, XtratuM unconditionally jumps to the entry point provided for the RSW in the system XML configuration file.

On the other hand, when this option is set, XtratuM calls the `UsrReset` function. This function must be provided by the user to XtratuM during its compilation process, being integrated within the hypervisor.

In order to do so, developer must define the `USR_RESET_CODE` variable pointing out to the file with the function. For instance:

```
$ export USR_RESET_CODE=/home/xmdeveloper/reset_func.c
$ make
```

If the function `UsrReset` is not provided, the XtratuM compilation fails with the following error message:

```
miguel@sneaker:core$ make
make[1]: Entering directory ‘/home/xmdeveloper/xm-q/user’
make[1]: Leaving directory ‘/home/xmdeveloper/xm-q/user’

> Building XM Core
  - kernel/sparcv8
  - kernel/mmu
```

```
  - kernel
  - klibc
  - klibc/sparcv8
  - objects
  - drivers
> Linking XM Core
kernel/kern.o: In function `ResetSystem':
/home/xmdeveloper/xm-q/core/kernel/setup.c:138: undefined reference to `UsrReset'
make: *** [core] Error 1
```

The `UsrReset` function defines the following signature:

```
...
/*
  These two headers must be included in order to define
  xmHmLog_t data structure
*/
#include <arch/paging.h>
#include <objects/hm.h>

...

void UsrReset(xmHmLog_t *log) {
  /* To be implemented */
  /*
     If log == NULL then
        -- Reset was caused by XM_syste_reset service
     else
        -- Reset occurred by a HM event.
     end if;
*/
}
```

Listing 2.1: UsrReset function implementation

Where `log` is **NULL** if the cold system reset was carried out by the `XM_reset_system()` hypercall or a valid HM log if it was caused as a result of a HM event.

Note that `UsrReset` should not return. To prevent an undefined behaviour if `UsrReset` does not honor this behaviour, XtratuM executes an endless loop just returning from this function.

The function is executed as part of the hypervisor code in the same conditions as the rest of the XtratuM code. That is, this function can access to any internal data or function of the hypervisor. The function is called with a valid stack and the MMU enabled.

## 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 applications[1]: 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.

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

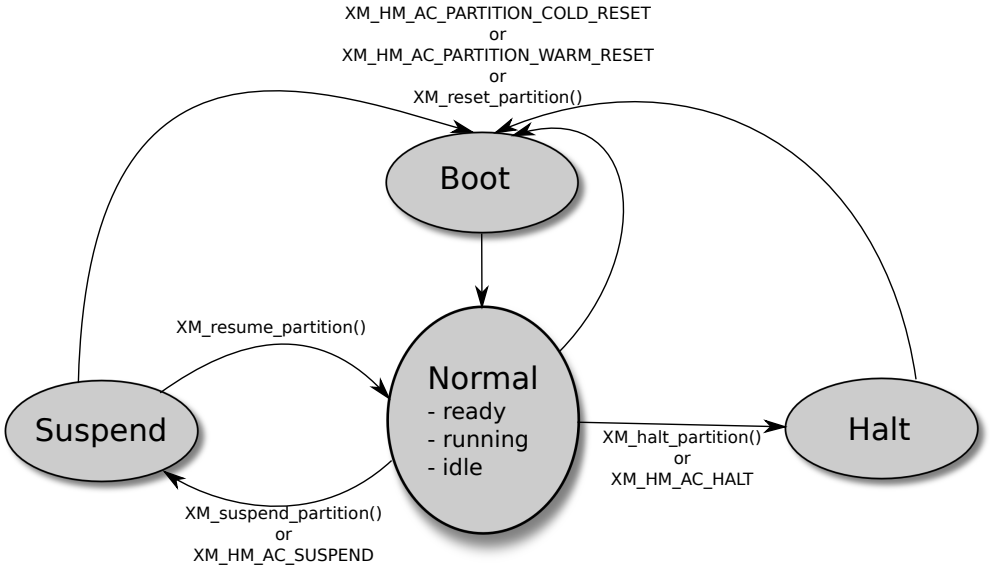



Figure 2.3: Partition states and transitions.

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()`).

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()` 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.

| Hypercall | System |
|---|---|
| XM_get_gid_by_name | Partial |
| XM_get_partition_status | Partial |
| XM_get_system_status | Yes |
| XM_halt_partition | Partial |
| XM_halt_system | Yes |
| XM_hm_read | Yes |
| XM_hm_status | Yes |
| XM_memory_copy | Partial |
| XM_reload_watchdog | Yes |
| XM_reset_partition | Partial |
| XM_reset_system | Yes |
| XM_resume_partition | Yes |
| XM_shutdown_partition | Partial |
| XM_suspend_partition | Partial |
| XM_switch_sched_plan | Yes |
| XM_trace_read | Yes |
| XM_trace_status | Yes |

Table 2.1: List of system reserved hypercalls.

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.

## 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 structures[2]. The fist "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]For efficiency and simplicity reasons.

## 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 aware of the scheduling policy used internally on each partition.

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 1 | 0 | 20 | 100 | 20 | | | | |
| Partition 2 | 20 | 10 | 120 | 10 | | | | |
| Partition 3 | 40 | 30 | 140 | 30 | | | | |
| Partition 4 | 30 | 10 | 70 | 10 | 130 | 10 | 170 | 10 |
| Partition 5 | 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.

```
<?xml version="1.0"?>
<Processor frequency="80Mhz" id="0">
  <Sched>
    <CyclicPlan>
      <Plan majorFrame="1s">
        <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"/>
        <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"/>
```

```
        <Slot duration="20ms" id="10" partitionId="0" start="180ms"/>
      </Plan>
    </CyclicPlan>
  </Sched>
</Processor>
```

Listing 2.2: Plan example

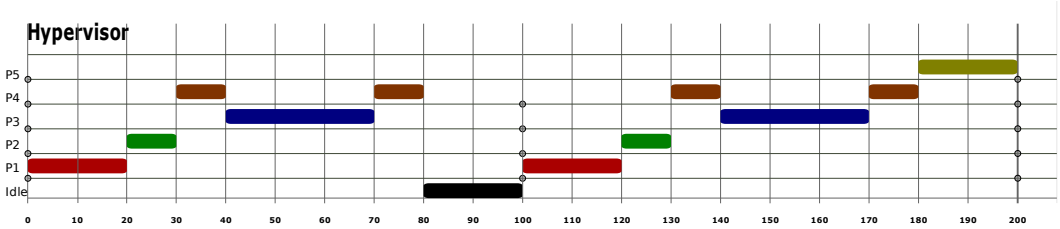



Figure 2.4: Scheduling example.

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. 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.

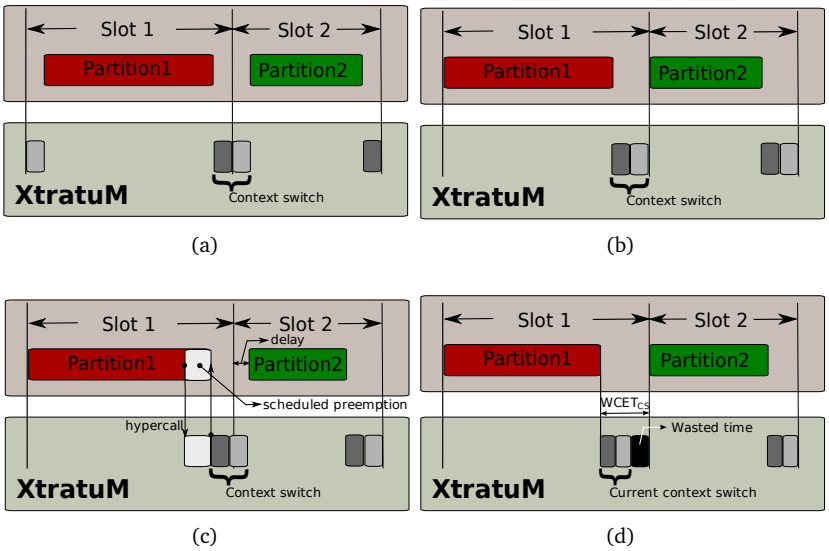



Figure 2.5: Xtratum context switch analysis.

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.

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.

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 ($BCET_{CS}$).
- 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.

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.

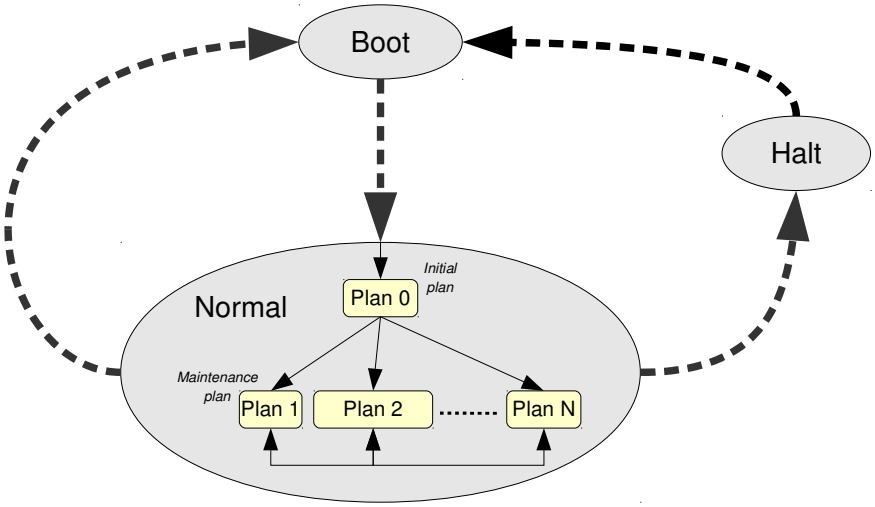



Figure 2.6: Scheduling modes.

- Requested from a system partition. The plan switch occurs at the end the current plan.

  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 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 a hypercall), 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.

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.

### 2.5.2 External synchronisation of the scheduling plan

XtratuM provides the capability to synchronise the scheduling plan with a periodic external signal. The *system integrator* defines an external periodic synchronisation signal that is used to re-execute the scheduling plan once it finishes. Any scheduling plan synchronised with this external signal must define a `MAF` multiple of the period of the signal. XtratuM also introduces a *tolerance synchronisation threshold* defining the interval $[MAF - syncThreshold, MAF + syncThreshold]$ where the arrival of the synchronisation signal is considered valid.

When the scheduling plan is waiting the synchronisation signal and it arrives outside of the synchronisation interval, it is considered a fail which raises the `XM_HM_EV_EXTSYNC_ERROR` event (see 2.9).

The plan first synchronisation requires an special consideration, XtratuM defines a valid interval of $[currenttime, MAF + syncThreshold]$ for waiting the synchronisation signal. If the signal does not arrive within this interval then the `XM_HM_EV_EXTSYNC_ERROR` event is raised.

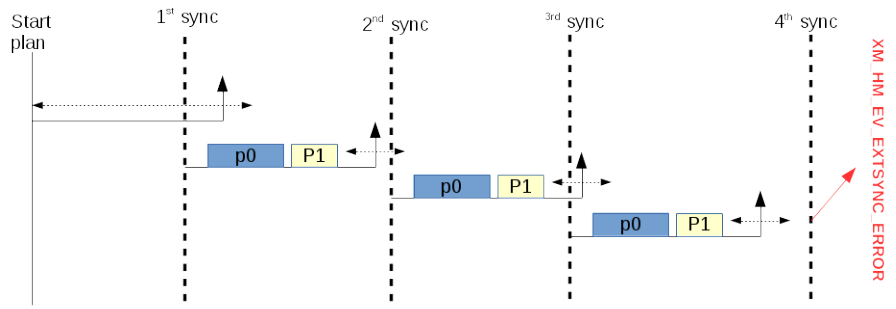



Figure 2.7: MAF synchronisation example

Figure 2.7 shows an example of plan being synchronised with an external signal. During the initial phase, the XtratuM scheduler waits until the synchronisation signal arrives to start the execution of the plan. Once this occurs, the plan is re-executed periodically with respect to the synchronisation signal until the fourth re-execution where the signal arrives outside the tolerance interval defined by the scheduler, causing a `XM_HM_EV_EXTSYNC_ERROR` event. Note that the plan starting point may be shifted with respect to the original plan execution, causing a temporal drift.

The execution of the plan starts defining a tolerance interval of $[currenttime, MAF+syncThreshold]$. The first time the synchronisation signal occurs within this interval so the plan starts its execution. The re-execution of the plan is performed correctly until the synchronisation signal (fourth time) arrives outside the tolerance interval, generating a `XM_HM_EV_EXTSYNC_ERROR`.

This feature is enabled by setting "`Enable external synchronisation`" option during the XtratuM configuration process (see 8.1).

Once enabled, the definition of the scheduling plan, `./Plan`, in the XML configuration file requires two new attributes:

- `@extSync`: defines the hardware interrupt used external synchronisation signal.
- `@interval`: defines the threshold used to define the synchronisation tolerance interval.

Note that on the case of defining multiple plans on a processor, if one plan requires an external synchronisation signal, all scheduling plans must be synchronised on the same external signal. Additionally, all these plans must define the same MAF duration, in concordance with the frequency of the expected external synchronisation signal.

## 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 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. See section 5.17 for more details.

## 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 responsability of the integrator to define the appropriated cache feature taking into account the system characteristics.

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.

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 variable[3] block of data. A message is sent from a source partition to one or more destination 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 the partition level, messages are atomic entities i.e., either the whole message is received or nothing is received. Partition developers are responsible for agreeing on the format (data types, endianess, 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.

[3] XtratuM defines the maximum length of a 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.

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

Sending and receiving messages are 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 receive 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).

## 2.9 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 the 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 a fault or confine an error.

**HM configuration:**
to bind the occurence of each HM event with the appropriate HM action.

**HM notification:**
to report the occurrence of the HM events.

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.

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:

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.

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).

The scope[6] where an HM event should be managed has to be greater than the scope where it is "believed" it can be produced.

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 when while XtratuM is executing; and partition level if the event is raised while a partition is running.

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

Note that although in the LEON2 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.

[4] Fault: What is believed to be the original reason that caused an error.
[5] Error: The manifestation of a fault.
[6] The term **level** is used in the ARINC-653 standard to refer to this idea

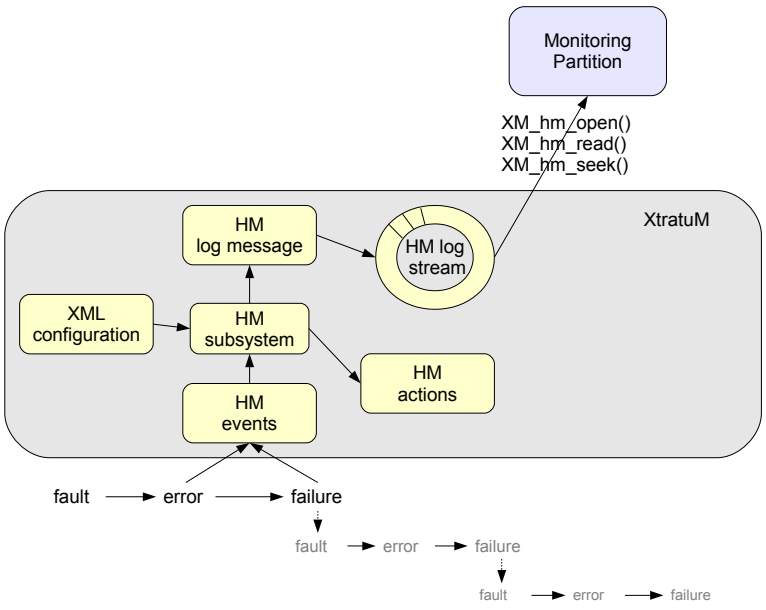



Figure 2.8: Health monitoring overview.

### 2.9.1 HM Events

There are three sources of HM events:

- Events caused by abnormal hardware behaviour. These events are notified to XtratuM via processor traps. 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. These events are defined as application error events.

- 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 id, memory address, faulting device, etc.) is gathered and used to select the apropiate HM action.

The list of HM events and their respective numeric id is:

| Action | Numeric id |
|---|---|
| XM_HM_EV_UNEXPECTED_TRAP | 1 |
| XM_HM_EV_PARTITION_UNRECOVERABLE | 2 |
| XM_HM_EV_PARTITION_ERROR | 3 |
| XM_HM_EV_PARTITION_INTEGRITY | 4 |
| XM_HM_EV_MEM_PROTECTION | 5 |
| XM_HM_EV_OVERRUN | 6 |
| XM_HM_EV_SCHED_ERROR | 7 |
| XM_HM_EV_WATCHDOG_TIMER | 8 |
| XM_HM_EV_INCOMPATIBLE_INTERFACE | 9 |
| XM_HM_EV_EXTSYNC_ERROR | 10 |
| XM_HM_EV_SPARCV8_WRITE_ERROR | 11 |
| XM_HM_EV_SPARCV8_INSTR_ACCESS_MMU_MISS | 12 |
| XM_HM_EV_SPARCV8_INSTR_ACCESS_ERROR | 13 |
| XM_HM_EV_SPARCV8_REGISTER_HARDWARE_ERROR | 14 |
| XM_HM_EV_SPARCV8_INSTR_ACCESS_EXCEPTION | 15 |
| XM_HM_EV_SPARCV8_PRIVILEGED_INSTR | 16 |
| XM_HM_EV_SPARCV8_ILLEGAL_INSTR | 17 |
| XM_HM_EV_SPARCV8_FP_DISABLED | 18 |
| XM_HM_EV_SPARCV8_CP_DISABLED | 19 |
| XM_HM_EV_SPARCV8_UNIMPLEMENTED_FLUSH | 20 |
| XM_HM_EV_SPARCV8_WATCHPOINT_DETECTED | 21 |
| XM_HM_EV_SPARCV8_MEM_ADDR_NOT_ALIGNED | 22 |
| XM_HM_EV_SPARCV8_FP_EXCEPTION | 23 |
| XM_HM_EV_SPARCV8_CP_EXCEPTION | 24 |
| XM_HM_EV_SPARCV8_DATA_ACCESS_ERROR | 25 |
| XM_HM_EV_SPARCV8_DATA_ACCESS_MMU_MISS | 26 |
| XM_HM_EV_SPARCV8_DATA_ACCESS_EXCEPTION | 27 |
| XM_HM_EV_SPARCV8_TAG_OVERFLOW | 28 |
| XM_HM_EV_SPARCV8_DIVIDE_EXCEPTION | 29 |
| XM_HM_EV_SPARCV8_UNCORRECTABLE_EDAC_ERROR | 30 |
| XM_HM_EV_APP_DEADLINE_MISSED | 31 |
| XM_HM_EV_APP_APPLICATION_ERROR | 32 |
| XM_HM_EV_APP_NUMERIC_ERROR | 33 |
| XM_HM_EV_APP_ILLEGAL_REQUEST | 34 |
| XM_HM_EV_APP_STACK_OVERFLOW | 35 |
| XM_HM_EV_APP_MEMORY_VIOLATION | 36 |
| XM_HM_EV_APP_HARDWARE_FAULT | 37 |
| XM_HM_EV_APP_POWER_FAIL | 38 |

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

### 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 system or hypervisor scope;

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

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 the 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.

In the case that the logs are stored in a log stream, then they can be retrieved by system partitions 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_u16_t signature;
//   xm_u16_t checksum;
    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
#define HMLOG_OPCODE_SYS_MASK (0x1<<HMLOG_OPCODE_SYS_BIT)
#define HMLOG_OPCODE_SYS_BIT 16
```

```
#define HMLOG_OPCODE_VALID_CPUCTXT_MASK (0x1<<
    HMLOG_OPCODE_VALID_CPUCTXT_BIT)
#define HMLOG_OPCODE_VALID_CPUCTXT_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)
#define HMLOG_OPCODE_PARTID_BIT 0
//    xm_u32_t seq;
    xmTime_t timestamp;
    union {
#define XM_HMLOG_PAYLOAD_LENGTH 4
        struct hmCpuCtxt cpuCtxt;
        xmWord_t payload[XM_HMLOG_PAYLOAD_LENGTH];
    };
} __PACKED;

typedef struct xmHmLog xmHmLog_t;
```

Listing 2.3: 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, otherwise clear.

**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 of when the event was detected.

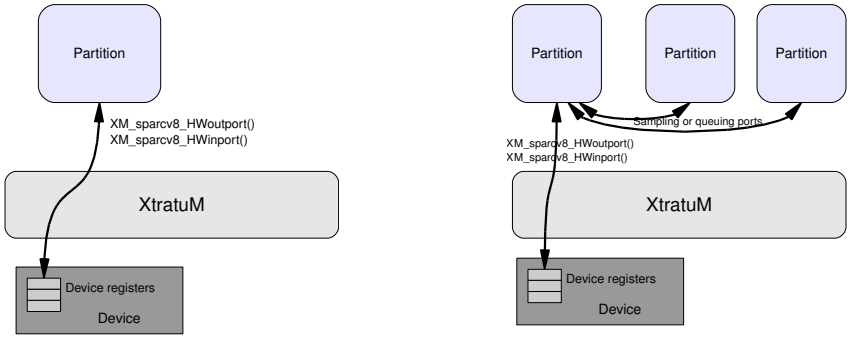
**Either cpuCtxt or payload:** When validCpuCtxt bit is set (opCode), then this field holds the processor context when the HM event was generated, otherwise, this field may hold information of the generated event. For instance, an application can manually generate a HM event specifying this information.

## 2.10 Access to devices

A partition, using exclusively a device (peripheral), can access the device through the device driver implemented in the partition (figure 2.9a). The partition is in charge of handling properly the device. The configuration file has to specify the I/O ports and the interrupt lines that will be used by each partition.

Two partitions cannot use the same interrupt line. XtratuM provides a fine grain access control to I/O ports, so that, several partitions can use (read and write) different bits of the the same I/O port. Also, it is possible to define a range of valid values that can be written in an I/O port (see section 5.10).

When a device is used by several partitions, an 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




(a) Partition using exclusively a device.

(b) I/O Server partition.

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.

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.

## 2.11 Traps, interrupts and exceptions

### 2.11.1 Traps

A **trap** is the mechanism provided by the SPARCv8 processor to implement the asynchronous transfer of control. When a trap occurs, the processor switches to supervisor mode and unconditionally jumps into a predefined handler.

SPARC v8 defines 256 different trap handlers. The table which contains these handlers is called *trap table*. The address of the trap table is stored in a special processor register (called `$tbr`). Both, the `$tbr` and the contents of the trap table are exclusively managed by XtratuM. All native traps jump into XtratuM routines.

The trap mechanism is used for several purposes:

**Hardware interrupts** Used by peripherals to request the attention of the processor.

**Software traps** Raised by a processor instruction; commonly used to implement the system call mechanism in the operating systems.

**Processor exceptions** Raised by the processor to inform about a condition that prevents the execution of an instruction. There are basically two kind of exceptions: those caused by the normal operation of the processor (such as register window under/overflow), and those caused by an abnormal situation (such as a memory error).

XtratuM defines 32 new interrupts called *extended interrupts*. These new interrupts are used to inform the partition about XtratuM specific events. In the case of SPARC v8, these interrupts are vectored starting at trap handler 224.

The trap handler raised by a trap can be changed by invoking the `XM_route_irq()` hypercall.

Partitions are not allowed to directly access (read or write) the `$tbr` register. XtratuM implements a *virtual trap table* instead.

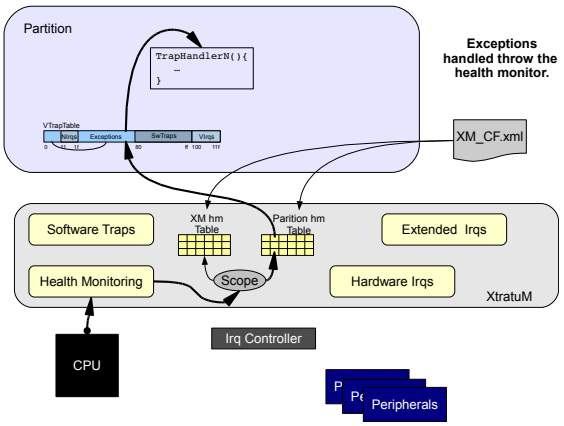



Figure 2.9: Exceptions handled by the health monitoring subsystem.

### 2.11.2 Interrupts

Although in a fully virtualised environment, a partition should not need to manage hardware interrupts; XtratuM only virtualises those hardware peripherals that may endanger the isolation, but leaves to the partitions to directly manage non-critical devices.

In order to properly manage peripherals, a partition needs to:

1. have access to the peripheral control and data registers.
2. be informed about triggered interrupts.
3. be able to block (mask and unmask) the associated interrupt line.

A hardware interrupt can only be allocated to one partition (in the `XM_CF` configuration file). The partition can then mask and unmask the hardware line in the native interrupt controller by using the `XM_mask_irq()` and `XM_unmask_irq()` functions.

XtratuM extends the concept of processor traps by adding 32 additional interrupts. This new range is used to inform the partition about events detected or generated by XtratuM.

Figure 2.10 shows the sequence from the occurrence of an interrupt to the partition's trap handler.

Partitions shall manage this new set of events in the same way standard traps are. These new interrupts are vectored in the trap table. These trap handlers are invoked by XtratuM on the occurrence of an event alike a standard SPARCv8 trap.

## 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.

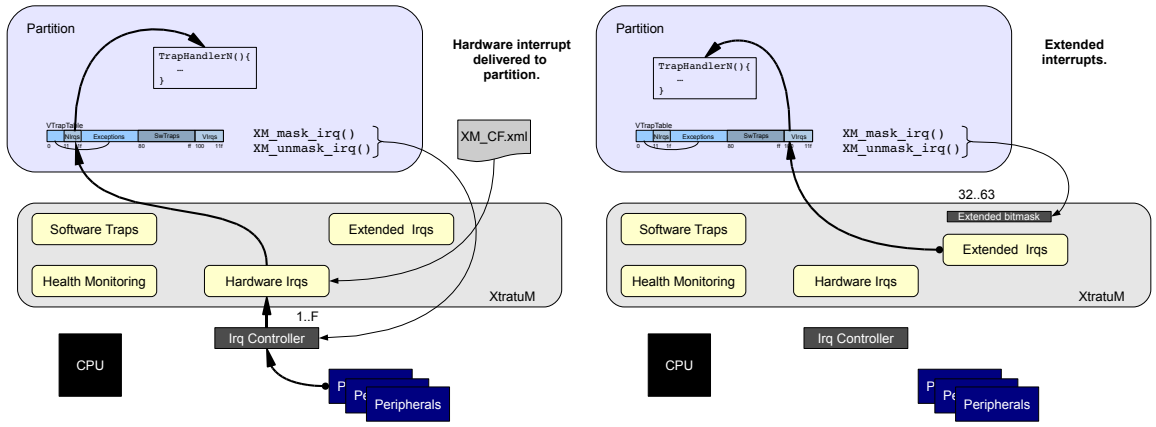



Figure 2.10: Hardware and extended interrupts delivery.

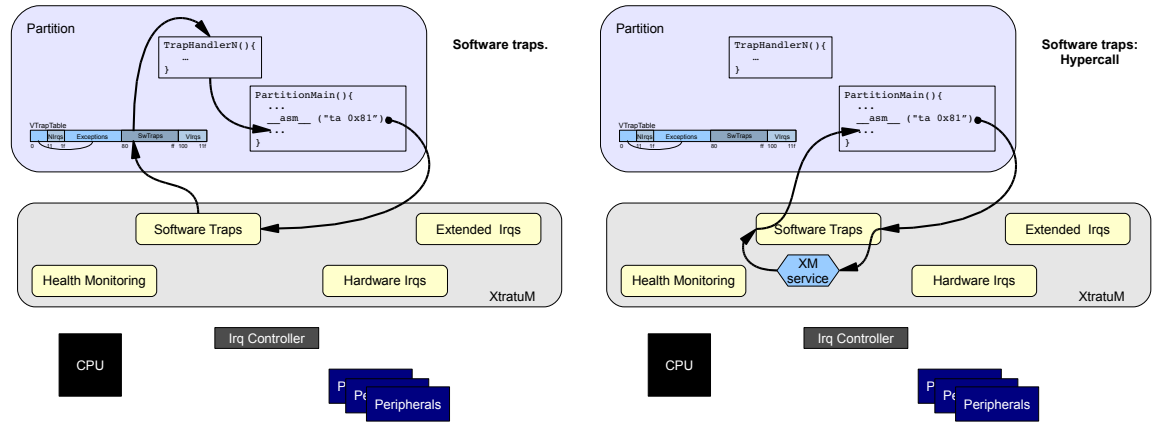



Figure 2.11: Software traps.

## 2.13 Clocks and timers

There are two clocks per partition:

**XM_HW_CLOCK:** Associated with the native hardware clock. The resolution is 1$\mu$sec.

**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 1$\mu$sec.

Only one timer can be armed for each clock.

## 2.14 Asymmetric Multi-Processing architecture

Asymmetric Multi-Processing software architectures (AMP) are supported by allowing running multiple instances of XtratuM in the same board.

In order to keep full compatibility with the already existing architecture, a new tool, `ampbuilder.py`, has been implemented. This tool packs as many XtratuM-based systems as CPUs are present on the board, additionally, this tool, includes a boot loader, which sets up each CPU to run a different instance of XtratuM.

Each XtratuM system must be configured in order to avoid hardware allocation overlapping. For instance, allocating the same UART port to both XtratuM will likely result in an incoherent output.

Internally, devices such as timers and the interrupt controller are allocated to each instance of XtratuM according the CPU id which is running it. This allocation consists in:

- Timers allocation: an instance of XtratuM requires two timers, one to implement a clock and another as a timer. In the case of the board bi-core GR712RC, timers 1 and 2 are allocated to CPU0, and timers 3 and GRTIMER 1 are allocated to CPU1.
- IRQ controller: each CPU has a different register to manage its IRQs (masking, forcing, cleaning, etc). Each instance is aware of the CPU over it is being executed; this information is required by XtratuM in order to manage correctly its IRQ controller.

Queuing and sampling channels have been adapted in order to allow a transparent communication, no matters the instance of XtratuM which is executing the partition. This adaptation consist in explicitly defining the address and the size which must be used by a channel. Inside the channel, external ports (ports belonging to a different XtratuM instance must be tagged as `external`). Internally, XtratuM implements spin-locks and IPIs to synchronise the use of this shared memory address.

XM0 configuration example:

```
...
  <PartitionTable>
    <Partition id="0" name="Partition1" flags="system boot" console="Uart">
      <PhysicalMemoryAreas>
        <Area start="0x40040000" size="256KB"/>
      </PhysicalMemoryAreas>
      <TemporalRequirements duration="500ms" period="500ms"/>
      <PortTable>
        <Port name="port1" type="sampling" direction="source"/>
        <Port name="port2" type="queuing" direction="source"/>
      </PortTable>
    </Partition>
  </PartitionTable>
  <Channels>
    <SamplingChannel maxMessageLength="64B" address="0x40080000" size="264KB">
      <Source portName="port1" partitionId="0" />
      <ExternalDestination portName="port1" cpuId="1" />
    </SamplingChannel>
    <QueuingChannel maxNoMessages="4" maxMessageLength="64B"
       address="0x40080000" size="64KB">
      <Source portName="port2" partitionId="0" />
      <ExternalDestination portName="port2" cpuId="1" />
    </QueuingChannel>
   </Channels>
...
```

XM1 configuration file:

```
...
```

```
  <PartitionTable>
    <Partition id="0" name="Partition1" flags="system boot" console="Uart">
      <PhysicalMemoryAreas>
        <Area start="0x60180000" size="256KB"/>
      </PhysicalMemoryAreas>
      <TemporalRequirements duration="500ms" period="500ms"/>
      <PortTable>
        <Port name="port1" type="sampling" direction="destination"/>
        <Port name="port2" type="queuing" direction="destination"/>
      </PortTable>
    </Partition>
  </PartitionTable>
  <Channels>
    <SamplingChannel maxMessageLength="64B" address="0x40080000" size="264KB">
      <ExternalSource portName="port1" cpuId="0" />
      <Destination portName="port1" partitionId="0" />
    </SamplingChannel>
    <QueuingChannel maxNoMessages="4" maxMessageLength="64B"
        address="0x40080000" size="64KB">
      <ExternalSource portName="port2" cpuId="0" />
      <Destination portName="port2" partitionId="0" />
    </QueuingChannel>
   </Channels>
...
```

These new attributes (`address` and `size`) are optional, enabling compatibility with pre-existing systems.

An AMP system can be built following the next steps:

1. Building an instance of a system (resident_sw0).
2. Building another instance of a system (resident_sw1).
3. Creating an image packing both instances through the `ampbuilder.py` tool.

## 2.15 Symmetric Multi-Processing (SMP) architecture

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

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 syncronisation point to start partition's scheduling.

The hardware timer GRTIMER1 is used by XtratuM as the timer for the CPU1. The timers allocated for implement the clock and timer for the CPU0 are the same used in the mono-core version: the Timer 0 and Timer 1 of the General Purpose Timer Unit.

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.

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.

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 parallely in both partitions.

```
...
<ProcessorTable>
  <Processor id="0" frequency="80Mhz">
    <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="80Mhz">
    <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>
...
```

## 2.16 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.

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

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.

## 2.17 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`.

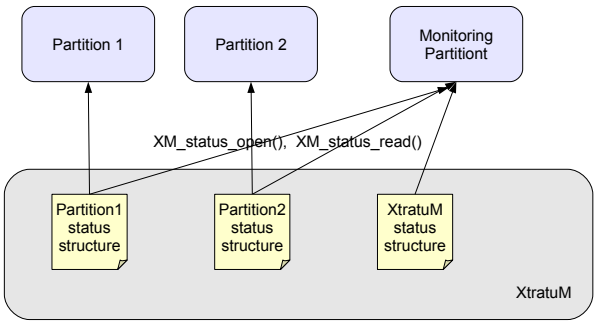



Figure 2.12: Status overview.

## 2.18 Drivers

### 2.18.1 MIL-STD-1553 device support

XtratuM provides a MIL-STD-1553 device driver that, if selected during the XtratuM configuration process (see section 8.1), initialises the device:

- Resets the device.

- Inits the device.
- Enables MIL-STD-1553. The clock source is configured. This device uses the hardware interrupt 14.
- Inits the device's descriptor table.

Once initialised, the device can be directly used by a partition by mapping the device I/O registers in the partition's memory area.

The MIL-STD-1553 driver is enabled by setting ``Enable MIL-STD-1553 as MAF Synchronisation'' option during the XtratuM configuration process. Once enabled, the MIL-STD-1553 driver requires the ``Remove Terminal Address'' and the ``Descriptor Memory Address'' to be configured (see section 8.1). The clock source used by the MIL-STD-1553 is also configurable.

## 2.19 Summary

Next 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 interrupts enabled. This behaviour is enforced by XtratuM.
- Partition's code is not allowed to disable native interrupts, only their own 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 kind of partitions: system and standard.
- Only system partitions are allowed to control the state of the system and other partitions, and to query about them.
- 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.

This page is intentionally left blank.

# Chapter 3

# Developing 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.

The simplest scenario is composed of two actors: the *integrator* and two *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).

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.
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,
   - design the scheduling plan,
   - communication ports between partitions,
   - the virtual devices and physical peripherals allocated to each partition,
   - configure the health monitoring,
   - etc.

   By creating the `XM_CF` configuration file[1]. See section 8.3 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:

[1] Although it is not mandatory to name "`XM_CF`" the configuration file, we will use this name in what follows for simplicity.

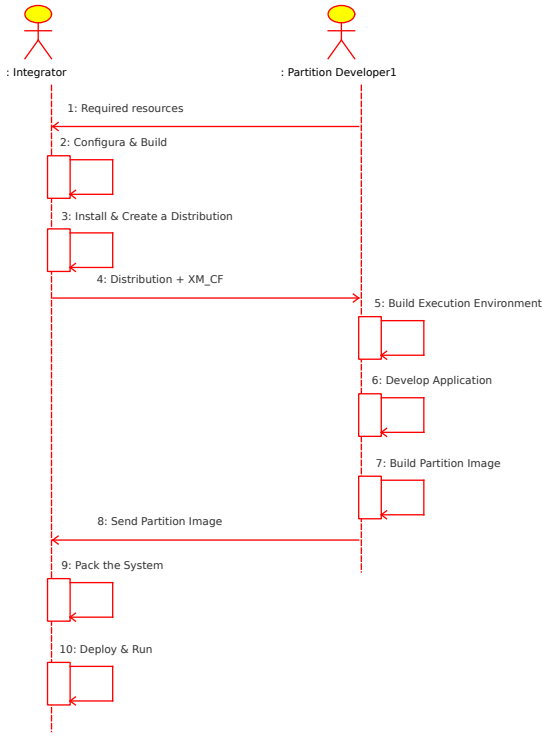



Figure 3.1: Integrator and partition developer interactions.

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 buid 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.

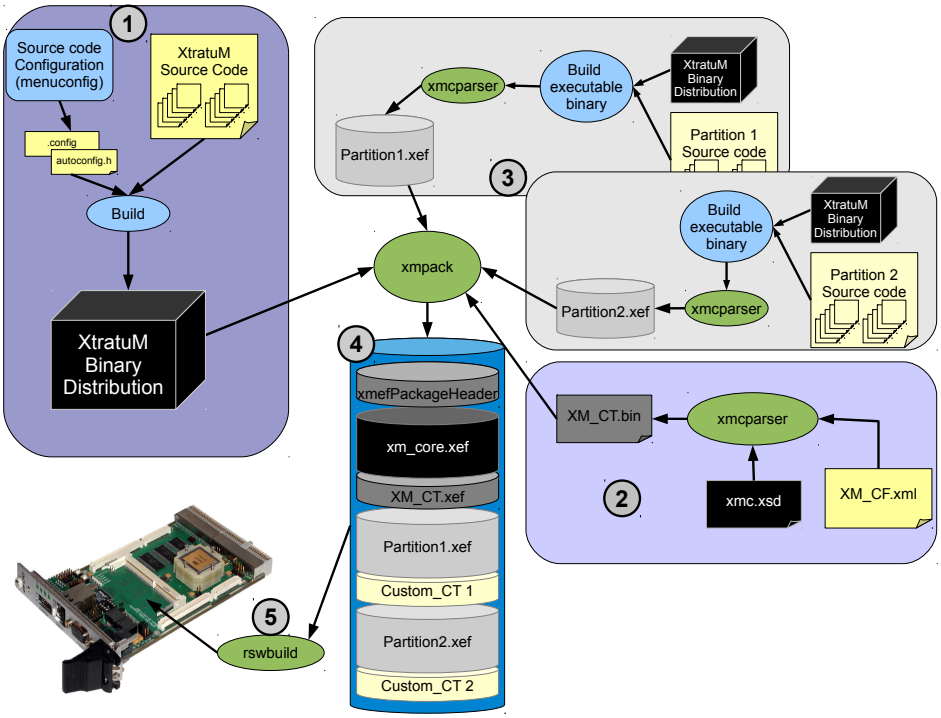



Figure 3.2: The big picture of building a XtratuM system.

`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.

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.

## 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

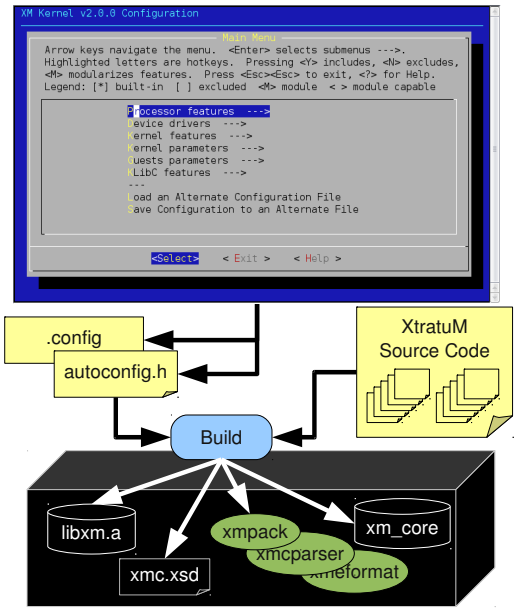



Figure 3.3: Menuconfig process.

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).

915 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 to be use "C" programs and the in Makefiles respectively.

920 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 synchronised.

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.

925 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 TGZ[2] package with all the files to develop with XtratuM, which can be delivered to the partition developers. See chapter 4.

## 3.3 System configuration

930 The integrator, jointly 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 8.3. Figure 3.4 shows a graphical view of the configuration schema.

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

[2]TGZ: 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.

**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.

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.

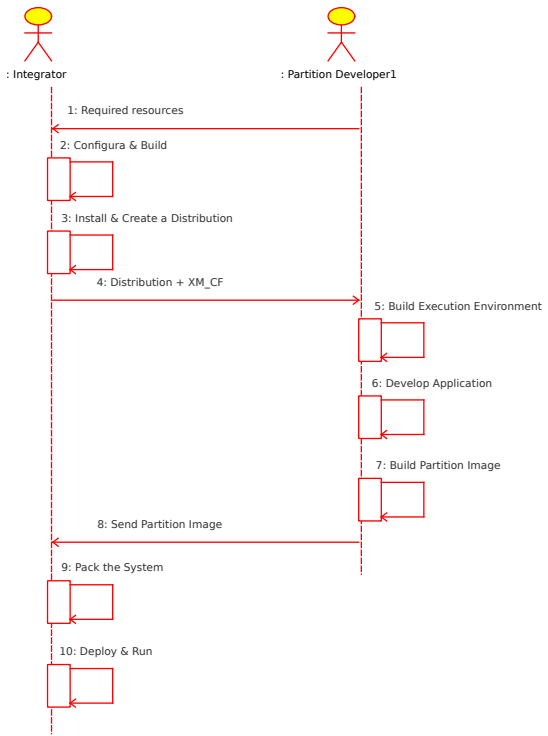



Figure 3.4: Graphical representation of an XML configuration file.

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 /refxmcparser.

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

## 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 require the following files:

**`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.

**`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`:** For converting an ELF file into an XEF one.

**`xmcparser`:** The tool to translate the configuration file (`XM_CF.xml`) into a "C" file which sould be compiled to produce the configuration table (`XM_CT`).

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.

## 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.

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.

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.

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 `rswbuild` creates an bootable ELF file with the resident software and the container.

This page is intentionally left blank.

# Chapter 4

# Building XtratuM

## 4.1 Developing environment

XtratuM has been compiled and tested with the following package versions:

| Package | Version | Linux package name | | Purpose |
|---|---|---|---|---|
| host gcc | 4.6.3 / 4.8.2 | gcc | req | Build host utilities |
| make | 3.81 | make | req | Core |
| binutils | 2.22 / 2.24 | binutils | req | Core |
| sparc-toolchain | linux-3.4.4 | | req | Core |
| libncurses | 5.9.20110404 / 5.9.20140118 | libncurses5-dev | req | Configure source code |
| libxml2 | 2.7.8 / 2.9.1 | libxml2-dev | req | Configuration parser |
| tsim | 2.0.37 | | opt | Simulated run |
| grmon | 1.1.52 | | opt | Deploy and run |
| perl | 5.14.2 / 5.18.2 | perl | opt | Testing |
| python | 2.6.9 / 2.7.6 / 3.4.0 | python | req | Compilation data preprocessing |
| makeself | 2.1.5 / 2.2.0 | makeself | opt | Build self extracting distribution |

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.

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

1. Check that the GNU LIBC Linux GCC 3.4.4 toolchain for SPARC LEON is installed in the system. It can be downloaded from: http://www.gaisler.com `sparc-linux-3.4.4-x.x.x.tar.bz2`

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.sparc` into `xmconfig`, and edit it to meet your system paths. The variable `XTRATUM_PATH` shall contain the root directory of XtratuM. Also,

if the sparc-linux toolchain directory is not in the PATH then the variable TARGET_CCPREFIX shall contain the path to the actual location of the corresponding tools. The prefix of the SPARC v8 tool chain binaries, shall be "sparc-linux-"; otherwise, edit also the appropiate variables.

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.sparc 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: The menuconfig target configures the XtratuM source code and the resident software. Therefore, two different configuration menus are presented, see section 8.1.

   For running XtratuM in the simulator, select the appropriate processor model from the menuconfig menus.

5. Compile XtratuM sources:

   ```
   $ make
   > Configuring and building the "XtratuM hypervisor"
   > Building XM Core
     - kernel/sparcv8
     - kernel/mmu
     - kernel
     - klibc
     - klibc/sparcv8
     - objects
     - drivers
   > Linking XM Core
      text    data     bss     dec     hex filename
     61949     152   81112  143213   22f6d xm_core
   de1daa3d40e5a2171d0b16c6fbbf3c6e xm_core.xef
   > Done

   > Configuring and building the "User utilities"
   > Building XM user
     - libxm
     - libxef
     - tools
     - tools/xmpack
     - tools/xmcparser
     - tools/xmgcov
     - tools/xmbuildinfo
     - tools/rswbuild
     - tools/xmcbuild
     - tools/xef
     - xal
     - bootloaders/rsw
   > Done
   ```

## 4.3 Generating a binary distribution

The generated files from the compilation process are in source code directories. In order to distribute the compiled binary version of XtratuM to 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/xm-distro.21476/xtratum-3.1.2/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-3.1.2.tar.bz2"
> Done

> Generating self extracting binary distribution "xtratum-3.1.2.run"
> Done
```

The files `xtratum-x.x.x.tar.bz2` or `xtratum-x.x.x.run` contains 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.

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 SPARCv8 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.

## 4.4 Installing a binary distribution

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

The install script requires only two parameters:

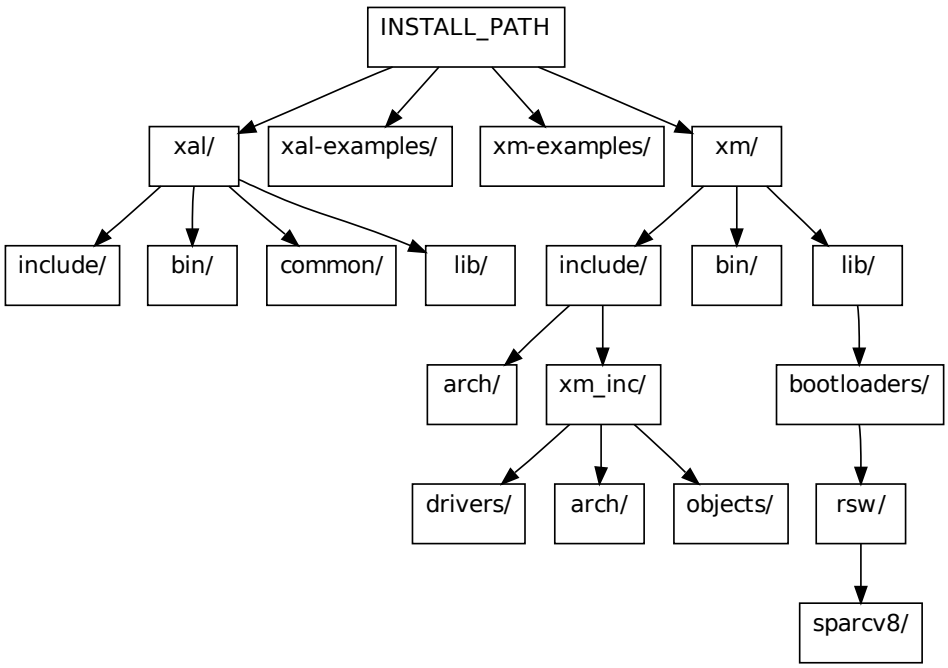



Figure 4.1: Content of the XtratuM distribution.

1. The installation path.
2. The path to the SPARC v8 toolchain.

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 SPARCv8 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-3.1.2.run
Verifying archive integrity... All good.
Uncompressing XtratuM binary distribution 3.1.2:......

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 sparc).
  The toolchain shall contain the executables named sparc-linux-*.

3. Confirm the installation settings.

Important: you need write permision 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/xm2env
2.- Path to the sparc toolchain [/opt/sparc-linux-3.4.4/bin/]: /opt/sparc-linux/bin

Confirm the Installation settings:
Selected installation path : /home/xmuser/xm3env
Selected toolchain path : /opt/sparc-linux/bin
```

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

Installation completed.
```

Listing 4.1: 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.
2. Compile the partition:

```
$ make
.....
Created by "xmuser" on "myhost" at "Wed Jul 20 10:11:32 CET 2011"
XM path: " /home/xmuser/xm2env/xm"

XtratuM Core:
   Version: "3.1.2"
   Arch:    "sparcv8"
   File:    "/home/xmuser/xm2env/xm/lib/xm_core.xef"
   Sha1:    "962b930e8278df873599e6b8bc4fcb939eb92a19"
   Changed: "2010-02-03 13:19:51.000000000 +0100"

XtratuM Library:
   Version: "3.1.2"
   File:    "/home/xmuser/xm2env/xm/lib/libxm.a"
   Sha1:    "46f64cf2510646833a320e1e4a8ce20e4cd4e0a9"
   Changed: "2010-02-03 13:19:51.000000000 +0100"

XtratuM Tools:
   File:    "/home/xmuser/xm2env/xm/bin/xmcparser"
   Sha1:    "8fa5dea03739cbb3436d24d5bc0e33b20906c47a"
```

   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".

3. To run this file just load it in the tsim or grmon and run (go) it:

```
$ tsim-leon3 -mmu
...
serial port A on stdin/stdout
allocated 4096 K RAM memory, in 1 bank(s)
allocated 16 M SDRAM memory, in 1 bank
allocated 2048 K ROM memory
icache: 1 * 4 kbytes, 16 bytes/line (4 kbytes total)
dcache: 1 * 4 kbytes, 16 bytes/line (4 kbytes total)
tsim> load resident_sw
section: .text, addr: 0x40200000, size 14340 bytes
section: .rodata, addr: 0x40203808, size 790 bytes
section: .container, addr: 0x40203b20, size 48888 bytes
section: .got, addr: 0x4020fa18, size 8 bytes
```

```
section: .eh_frame, addr: 0x4020fa20, size 64 bytes
read 38 symbols
tsim> go
resuming at 0x40200d24
XM Hypervisor (3.1 r2)
Detected 50.0MHz processor.
>> HWClocks [LEON clock (1000Khz)]
[CPU:0] >> HwTimer [LEON timer (1000Khz)]
[CPU:0] [sched] using cyclic scheduler
2 Partition(s) created
P0 ("Partition1":0) flags: [ SYSTEM BOOT (0x40080000) ]:
    [0x40080000 - 0x400fffff]
P1 ("Partition2":1) flags: [ SYSTEM BOOT (0x40100000) ]:
    [0x40100000 - 0x4017ffff]
Jumping to partition at 0x40082000
Jumping to partition at 0x40102000
I am Partition2
Hello World!
I am [CPU:0] [HYPERCALL] (0x1) Halted
Partition1
Hello World!
[CPU:0] [HYPERCALL] (0x0) Halted
```

## 4.6 XtratuM directory tree

```
sources
```

Listing 4.2: XtratuM sources tree

# Chapter 5

# Partition Programming

*This chapter explains how to build a XtratuM partition: partition developer tutorial.*

## 5.1 Implementation requirements

Below is a checklist of what the partition developer and the integrator should take into accout when using XtratuM. It is advisable to revisit this list to avoid incorrect assumptions.

**Development host:** 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.

Check that the executable files in `xm/bin` are compatible with the host architecture.

**Para-virtualised services:** Partition's code shall use the para-virtualised services. The use of native services is considered an error and the corresponding exception will be raised.

**PCT:** The Partition Control Table is located within the partition, mapped in read-only mode. Its content shall be modified by using the appropriate hypercalls.

**Store ordering:** XtratuM has been implemented considering that the LEON2 processor is operating in TSO (Total Store Ordering) mode. This is the standard SPARC v8 working mode. If changed to PSO (Partial Store Ordering) mode then random errors will happen.

**Memory allocation:**

- Care shall be taken to avoid overlapping the memory allocated to each partition.
- If MMU in not used, then the partition code shall be linked to work on the allocated memory areas. If the memory allocated in the `XM_CF` file is changed, then the linker script of the affected partition shall be updated accordingly.

**Reserved names:** The prefix "xm", both in upper and lower case, is reserved for XtratuM identifiers.

**Stack management:** XtratuM manages automatically the register window of the partitions. The partition code is responsible of initialising the stack pointer to a valid memory area, and reserve enough space to accommodate all the data that will be stored in the stack. Otherwise, a stack overflow may occur.

**Data Alignment:** By default, all data structures passed to or shared with XtratuM shall by aligned to 8 bytes.

**Units definition and abbreviations:**

**"KB"** (KByte or Kbyte) is equal to 1024 ($2^{10}$) bytes.

**"Kb"** (Kbit) is equal to 1024 ($2^{10}$) bits.

**"MB"** (MByte or Mbyte) is equal to 1048576 ($1024 \cdot 1024 = 2^{20}$) bytes.

**"Mb"** (Mbit) is equal to 1048576 ($1024 \cdot 1024 = 2^{20}$) bits.

**"Khz"** (Kilo Hertz) is equal to 1000 hertzs.

**"Mhz"** (Mega Hertz) is equal to 1000.000 hertzs.

**XtratuM memory footprint:** XtratuM does not use dynamic memory allocation. Therefore, all internal data structures are declared statically. The size of these data structures are defined during the source code configuration process.

The following configuration parameters are the ones the have an impact on the memory needed by XtratuM:

**Maximum identifier length:** Defines the space reserved to store the names of the partitions, ports, scheduling slots and channels.

**Kernel stack size:** For each partition, XtratuM reserves a kernel stack. Do not reduce the value of this parameter unless you know the implications.

**Partition memory areas (if the WPR is used and MMU disabled):** Due to the hardware device (WPR) used to force memory protection, the area of memory allocated to the partitions shall fulfil the next conditions:

- The size shall be greater than or equal to 32KB.
- The size shall be a power of two.
- The start address shall be a multiple of the size.

**Configuration of the resident software (RSW):** The information contained in the `XM_CF` regarding the RSW is not used to configure the RSW itself. That information is used:

- by XtratuM to perform a system cold reset,
- and by the `xmcparser` to check for memory overlaps.

**Partition declaration order:** The partition elements, in the `XM_CF` file, shall be ordered by "id" and the id's shall be consecutive starting at zero.

## 5.2 XAL development environment

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 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.

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
       `-- xm-examples
```

Listing 5.1: Installation tree.

The XAL subtree contains the following elements:

```
xal
     |-- bin                      # utilities
     |   |-- xpath
     |   `-- xpathstart
     |-- common                   # compilation rules
     |   |-- config.mk
     |   |-- config.mk.dist
     |   `-- rules.mk
     |-- include                  # headers
     |   |-- arch
     |   |   `-- irqs.h
     |   |-- assert.h
     |   |-- autoconf.h
     |   |-- config.h
     |   |-- ctype.h
     |   |-- irqs.h
     |   |-- limits.h
     |   |-- stdarg.h
     |   |-- stddef.h
     |   |-- stdio.h
     |   |-- stdlib.h
     |   |-- string.h
     |   `-- xal.h
     |-- lib                      # libraries
     |   |-- libxal.a
     |   `-- loader.lds
     `-- sha1sum.txt
```

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`.
- Define interrupt handlers and all services provided by XtratuM.

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.

In the xal-examples subtree, the reader can find several examples of XAL partitions and how these examples can be compiled. Next is shown the `Makefile` file.

```
# XAL_PATH: path to the XTRATUM directory
XAL_PATH=/......./xal

# XMLCF: path to the XML configuration file
XMLCF=xm_cf.sparcv8.xml

# PARTITIONS: partition files (xef format) composing the example
PARTITIONS=partition1.xef partition2.xef ....

all: container.bin resident_sw
include $(XAL_PATH)/common/rules.mk

partition1: dummy_xal.o
       $(LD) -o $@ $^ $(LDFLAGS) -Ttext=$(call xpathstart,1,$(XMLCF))
......

PACK_ARGS=-h $(XMCORE):xm_cf.xef.xmc \
       -p 0:partition1.xef\
       -p 1:partition2.xef\
       .....

container.bin: $(PARTITIONS) xm_cf.xef.xmc
       $(XMPACK) check xm_cf.xef.xmc $(PACK_ARGS)
       $(XMPACK) build $(PACK_ARGS) $@
       @exec echo -en "> Done [container]\n"
```

Listing 5.4: Makefile.

## 5.3 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:

- 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 interrupts (enable/disable interrupts) which have to be replaced by hypercalls[1] to ask for the hypervisor to enable/disable the interrupts.

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

**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 it has to be aware about it.

**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).

## 5.4 The “Hello World” example

Let’s start with a simple code that is not ready to be executed on XtratuM and needs to be adapted.

```
void main() {
    int counter =0;

    xprintf(‘‘Hello World!\n’’);
    while(1) {
        counter++;
        counter %= 100000;
    }
}
```

Listing 5.5: Simple example.

The first step is to initialise the virtual execution environment and call the entry point (`PartitionMain` in the examples) of the partition. The following files are provided as an example of how to build the partition image and initialise the virtual machine.

**`boot.S`:** The assembly code where the headers and the entry point are defined.

**`traps.c`:** Required data structures: PCT and trap handlers.

**`std_c.c, std_c.h`:** Minimal “C” support as `mencpy`, `xprintf`, etc.

**`loader.lds`:** The linker script that arranges the sections to build the partition image layout.

The `boot.S` file:

```
#include <xm.h>
#include <xm_inc/arch/asm_offsets.h>
//#include <xm_inc/hypercalls.h>

#define STACK_SIZE 8192
#define MIN_STACK_FRAME 0x60

.text
.align 8

.global start, _start

_start:
```

[1] para-virtualised operations provided by the hypervisor

```
start:
        /* Zero out our BSS section.
             */
        set _sbss, %o0
        set _ebss, %o1

1:
        st %g0, [%o0]
        subcc %o0, %o1, %g0
        bl 1b
        add %o0, 0x4, %o0

        set __stack_top, %fp

        mov %g1, %o0
        call init_libxm
        sub %fp, MIN_STACK_FRAME, %sp

        ! Set up TBR
        set sparc_write_tbr_nr, %o0
        set _traptab, %o1
        __XM_HC

        call PartitionMain
        sub %fp, MIN_STACK_FRAME, %sp

        set libXmParams, %o0
        ld    [%o0], %o1
        ld    [%o1 + 0x1c], %o1
        mov halt_partition_nr, %o0
        __XM_HC

/*        set ~0, %o1
        mov halt_partition_nr, %o0
        __XM_HC*/

1:      b 1b
        nop

ExceptionHandlerAsm:
        mov sparc_get_psr_nr, %o0
        __XM_AHC
        mov %o0, %l0
!       set sparc_flush_regwin_nr, %
    o0
!       __XM_AHC
        sub %fp, 48, %fp
        std %l0, [%fp+40]
        std %l2, [%fp+32]
        std %g6, [%fp+24]
        std %g4, [%fp+16]
        std %g2, [%fp+8]
        st %g1, [%fp+4]
        rd %y, %g5
        st %g5, [%fp]

        mov %l5, %o0
        call ExceptionHandler
        sub %fp, 0x80, %sp
        ld [%fp], %g1
        wr %g1, %y
        ld [%fp+4], %g1
        ldd [%fp+8], %g2
        ldd [%fp+16], %g4
        ldd [%fp+24], %g6
        ldd [%fp+32], %l2
        ldd [%fp+40], %l0
        add %fp, 48, %fp
        mov %l0, %o1
        mov sparc_set_psr_nr, %o0
        __XM_AHC
        set sparc_iret_nr, %o0
        __XM_AHC

ExtIrqHandlerAsm:
        mov sparc_get_psr_nr, %o0
        __XM_AHC
        mov %o0, %l0
!       set sparc_flush_regwin_nr, %
    o0
!       __XM_AHC
        sub %fp, 48, %fp
        std %l0, [%fp+40]
        std %l2, [%fp+32]
        std %g6, [%fp+24]
        std %g4, [%fp+16]
        std %g2, [%fp+8]
        st %g1, [%fp+4]
        rd %y, %g5
        st %g5, [%fp]
        mov %l5, %o0
        call ExtIrqHandler
        sub %fp, 0x80, %sp
        ld [%fp], %g1
        wr %g1, %y
        ld [%fp+4], %g1
        ldd [%fp+8], %g2
        ldd [%fp+16], %g4
        ldd [%fp+24], %g6
        ldd [%fp+32], %l2
        ldd [%fp+40], %l0
        add %fp, 48, %fp
        mov %l0, %o1
        mov sparc_set_psr_nr, %o0
        __XM_AHC
        set sparc_iret_nr, %o0
        __XM_AHC

HwIrqHandlerAsm:
        mov sparc_get_psr_nr, %o0
        __XM_AHC
        mov %o0, %l0
!       set sparc_flush_regwin_nr, %
    o0
!       __XM_AHC
        sub %fp, 48, %fp
        std %l0, [%fp+40]
        std %l2, [%fp+32]
        std %g6, [%fp+24]
        std %g4, [%fp+16]
        std %g2, [%fp+8]
        st %g1, [%fp+4]
        rd %y, %g5
        st %g5, [%fp]
        mov %l5, %o0
        call HwIrqHandler
        sub %fp, 0x80, %sp

        ld [%fp], %g1
        wr %g1, %y
        ld [%fp+4], %g1
        ldd [%fp+8], %g2
        ldd [%fp+16], %g4
        ldd [%fp+24], %g6
        ldd [%fp+32], %l2
        ldd [%fp+40], %l0
```

```
        add %fp, 48, %fp
        mov %l0, %o1
        restore
        save
        mov sparc_set_psr_nr, %o0
        __XM_AHC
        set sparc_iret_nr, %o0
        __XM_AHC

.data
.align 8
__stack:
        .fill (STACK_SIZE/4),4,0
__stack_top:

.previous


#define BUILD_IRQ(irqnr) \
        sethi %hi(HwIrqHandlerAsm), %
             l4 ;\
        jmpl %l4 + %lo(
             HwIrqHandlerAsm), %g0 ;\
        mov irqnr, %l5 ;\
        nop

#define BUILD_TRAP(trapnr) \
        sethi %hi(ExceptionHandlerAsm
             ), %l4 ;\
        jmpl %l4 + %lo(
             ExceptionHandlerAsm), %g0
              ;\
        mov trapnr, %l5 ;\
        nop             ;

#define BAD_TRAP(trapnr) \
1:      b 1b    ;\
        nop     ;\
        nop     ;\
        nop

#define SOFT_TRAP(trapnr) \
1:      b 1b    ;\
        nop     ;\
        nop     ;\
        nop

#define BUILD_EXTIRQ(trapnr) \
        sethi %hi(ExtIrqHandlerAsm),
             %l4 ;\
        jmpl %l4 + %lo(
             ExtIrqHandlerAsm), %g0 ;\
        mov (trapnr+224), %l5 ;\
        nop

.align 4096
_traptab:
! + 0x00: reset
t_reset: b start
        nop
        nop
        nop

! + 0x01:
    instruction_access_exception
        BUILD_TRAP(0x1)

! + 0x02: illegal_instruction
        BUILD_TRAP(0x2)

! + 0x03: privileged_instruction
        BUILD_TRAP(0x3)

! + 0x04: fp_disabled
        BUILD_TRAP(0x4)

! + 0x05: window_overflow
        BUILD_TRAP(0x5)


! ..........
```

Listing 5.6: user/examples/sparcv8/boot.S

The `__xmImageHdr` declares the required image header (see section 6) and one partition header[2]: `__xmPartitionHdr`.

The entry point of the partition (the first instruction executed) is labeled `start`. First off, the `bss` section is zeroed; the stack pointer (`%sp` register) is set to a valid address; the address of the partition header is passed to the libxm (`call InitLibxm`); the virtual trap table register is loaded with the direction of `__traptab`; and finally the user routine `PartitionMain` is called. If the main function returns, then an endless loop is executed.

The remaining of this file contains the trap handler routines. Note that the assembly routines are only provided as illustrative examples, and **should not be used on production application systems**. These trap routines just jump to “C” code which is located in the file `traps.c`:

```
#include <xm.h>
#include <xm_inc/arch/paging.h>
#include "std_c.h"

extern void start(void);
```

[2]Multiple partition headers can be declared to allocate several processors to a single partition (experimental feature not documented).

```
struct xmImageHdr xmImageHdr __XMIHDR = {
    .sSignature=XMEF_PARTITION_MAGIC,
    .compilationXmAbiVersion=XM_SET_VERSION(XM_ABI_VERSION, XM_ABI_SUBVERSION,
        XM_ABI_REVISION),
    .compilationXmApiVersion=XM_SET_VERSION(XM_API_VERSION, XM_API_SUBVERSION,
        XM_API_REVISION),
    .noCustomFiles=0,
#if 0
    .customFileTab={[0]=(struct xefCustomFile){
           .sAddr=(xmAddress_t)0x40105bc0,
           .size=0,
       },
    },
#endif
    .eSignature=XMEF_PARTITION_MAGIC,
};

void __attribute__ ((weak)) ExceptionHandler(xm_s32_t trapNr) {
    xprintf("exception 0x%x (%d)\n", trapNr, trapNr);
    //XM_halt_partition(XM_PARTITION_SELF);
}

void __attribute__ ((weak)) ExtIrqHandler(xm_s32_t trapNr) {
    xprintf("extIrq 0x%x (%d)\n", trapNr, trapNr);
//    XM_halt_partition(XM_PARTITION_SELF);
}

void __attribute__ ((weak)) HwIrqHandler(xm_s32_t trapNr) {
    xprintf("hwIrq 0x%x (%d)\n", trapNr, trapNr);
    // XM_halt_partition(XM_PARTITION_SELF);
}
```

Listing 5.7: user/examples/common/traps.c

Note that the "C" trap handler functions are defined as "`weak`". Therefore, if these symbols are defined elsewhere, the new declaration will replace this one.

The linker script that arranges all the ELF sections is:

```
/*OUTPUT_FORMAT("binary")*/
OUTPUT_FORMAT("elf32-sparc", "elf32-sparc", "elf32-sparc")
OUTPUT_ARCH(sparc)
ENTRY(start)

SECTIONS
{
  .text ALIGN (8): {
        . = ALIGN(4K);
        _sguest = .;
        *(.text.init)
        *(.text)
  }

  .rodata ALIGN (8) : {
        *(.rodata)
        *(.rodata.*)
        *(.rodata.*.*)
```

```
  }

  .data ALIGN (8) : {
        _sdata = .;
        *(.data)
        _edata = .;
  }

  .bss : {
        *(.bss.noinit)
        _sbss = .;
        *(COMMON)
        *(.bss)
        _ebss = .;
  }

  _eguest = .;

  /DISCARD/ :
  {
        *(.note)
        *(.comment*)
  }
}
```

Listing 5.8: user/examples/lib/loader.lds

The section `.text.ini`, which contains the headers, is located at the beginning of the file (as defined by the ABI). The section `.xm_ctl`, which contains the PCT table, is located at the start of the `bss` section to avoid being zeroed at the startup of the partition. The contents of these tables has been initialized by XtratuM before starting the partition. The symbols `_sguest` and `_eguest` mark the Start and End of the partition image.

The ported version of the previous simple code is the following:

```
#include "std_c.h"          /* Helper functions */
#include <xm.h>
void PartitionMain () {     /* ''C'' code entry point. */
    int counter=0;

    xprintf(''Hello World!\n'');
    while(1) {
        counter++;
        counter %= 100000;
    }
}
```

Listing 5.9: Ported simple example

Listing 5.10 shows the main compilation steps required to generate the final container file, which contains a complete XtratuM system, of a system of only one partition. The partition is only a single file, called `simple.c`. This example is provided only to illustrate the build process. It is advisable to use some of the Makefiles provided in the `xm-examples` (in the installed tree).

```
# --> Compile the partition souce code: [simple.c] -> [simple.o]
$ sparc-linux-gcc -Wall -O2 -nostdlib -nostdinc -Dsparcv8 -fno-strict-aliasing \
    -fomit-frame-pointer --include xm_inc/config.h --include xm_inc/arch/arch_types.h \
```

```
    -I[...]/libxm/include -DCONFIG_VERSION=2 -DCONFIG_SUBVERSION=1 \
    -DCONFIG_REVISION=3 -g -D_DEBUG_ -c -o simple.o simple.c

# --> Link it with the startup (libexamples.a)
$ sparc-linux-ld -o simple simple.o -n -u start -T[...]/lib/loader.lds -L../lib \
    -L[...]xm/lib --start-group `sparc-linux-gcc -print-libgcc-file-name ` -lxm -lxef \
    -lexamples --end-group -Ttext=0x40080000

# --> Convert the partition ELF to the XEF format.
$ xmeformat build -c simple -o simple.xef

# --> Compile the configuration file.
$ xmcparser -o xm_cf.bin.xmc xm_cf.sparcv8.xml

# --> Convert the configuration file to the XEF format.
$ xmeformat build -c -m xm_cf.bin.xmc -o xm_cf.xef.xmc

# --> Pack all the XEF files of the system into a single container
$ xmpack build -h [...]/xm/lib/xm_core.xef:xm_cf.xef.xmc -p 0:simple.xef container.bin

# --> Build the final bootable file with the resident sw and the container.
$ rswbuild container.bin resident_sw
```

Listing 5.10: Example of a compilation sequence.

The partition code shall be compiled with with the flags `-nostdlib` and `-nostdinc` to avoid using host specific facilities which are not provided by XtratuM. The bindings between assembly and "C" are done considering that not frame pointer is used: `-fomit-frame-pointer`.

All the object files (`traps.o`,`boot.o` and `simple.o`) are linked together, and the `text` section is positioned in the direction `0x40050000`. This address shall be the same than the one declared in the `XM_CF` file:

```
<SystemDescription xmlns="http://www.xtratum.org/xm-3.x" version="1.0.0" name="
    Example">
      <HwDescription>
        <MemoryLayout>
          <Region type="rom" start="0x0" size="4MB" />
          <Region type="stram" start="0x40000000" size="4MB"/>
          <Region type="sdram" start="0x60000000" size="1MB"/>
        </MemoryLayout>
        <ProcessorTable>
          <Processor id="0" frequency="50Mhz">
            <CyclicPlanTable>
              <Plan id="0" majorFrame="2ms">
                <Slot id="0" start="0ms" duration="1ms" partitionId="0"/>
                <Slot id="1" start="1ms" duration="1ms" partitionId="1"/>
              </Plan>
            </CyclicPlanTable>
          </Processor>
        </ProcessorTable>
        <Devices>
          <Uart id="0" baudRate="115200" name="Uart"/>
          <MemoryBlock name="MemDisk0" start="0x4000" size="256KB" />
          <!--
            <MemoryBlock name="MemDisk0" start="0x40100000" size="256KB"/>
            <MemoryBlock name="MemDisk1" start="0x40150000" size="256KB"/>
            <MemoryBlock name="MemDisk2" start="0x40200000" size="256KB"/>
            -->
```

```
        </Devices>
      </HwDescription>
      <XMHypervisor console="Uart">
        <PhysicalMemoryArea size="512KB"/>
      </XMHypervisor>

      <!-- <track id="hello-xml-sample">-->
      <PartitionTable>
        <Partition id="0" name="Partition1" flags="system" console="Uart">
          <PhysicalMemoryAreas>
            <Area start="0x40080000" size="512KB"/>
          </PhysicalMemoryAreas>
          <TemporalRequirements duration="500ms" period="500ms"/>
        </Partition>
        <Partition id="1" name="Partition2" flags="system" console="Uart">
          <PhysicalMemoryAreas>
            <Area start="0x40100000" size="512KB" flags=""/>
          </PhysicalMemoryAreas>
          <TemporalRequirements duration="500ms" period="500ms"/>
          <HwResources>
            <Interrupts lines="4"/>
          </HwResources>
        </Partition>
      </PartitionTable>
</SystemDescription>
```

Listing 5.11: XML configuration file example

In order to avoid inconsistences between the memory `@Area` attribute of the configuration and the parameter passed to the linker, the `examples/common/xpath` tool[3] can be used, from a Makefile, to extract the information from the configuration file.

```
$ cd user/examples/hello_world
$ ../common/xpath -c -f xm_cf.sparcv8.xml /SystemDescription/PartitionTable/
    Partition[1]/PhysicalMemoryAreas/Area[1]/@start
0x40050000
```

Listing 5.12: Using xpath to recover to memory area of the first partition.

The attribute `/SystemDescription/PartitionTable/Partition[1]/PhysicalMemoryAreas/Area[1]/-@start` is the xpath reference to the attribute which defines the first region of memory allocated to the first partition, which in the example is the place where the partition will be loaded.

### 5.4.1 Included headers

The include header which contains all the definitions and declarations of the `libxm.a` library is **`xm.h`**. This file depends (includes) also the next list of files:

```
user/libxm/include/xm.h
user/libxm/include/xm_inc/config.h
user/libxm/include/xm_inc/autoconf.h
user/libxm/include/xm_inc/arch/arch_types.h
user/libxm/include/xm_inc/xmef.h
user/libxm/include/xm_inc/linkage.h
user/libxm/include/xm_inc/arch/linkage.h
user/libxm/include/xm_inc/arch/paging.h
user/libxm/include/xmhypercalls.h
```

[3] `xpath` is a small shell script frontend to the `xmllint` utility.

```
user/libxm/include/xm_inc/hypercalls.h
user/libxm/include/xm_inc/arch/hypercalls.h
user/libxm/include/xm_inc/arch/irqs.h
user/libxm/include/xm_inc/arch/xm_def.h
user/libxm/include/xm_inc/arch/leon.h
user/libxm/include/arch/xmhypercalls.h
user/libxm/include/xm_inc/objdir.h
user/libxm/include/xm_inc/guest.h
user/libxm/include/xm_inc/arch/atomic.h
user/libxm/include/xm_inc/arch/guest.h
user/libxm/include/xm_inc/arch/processor.h
user/libxm/include/xm_inc/xmconf.h
user/libxm/include/xm_inc/arch/xmconf.h
user/libxm/include/xm_inc/devid.h
user/libxm/include/xm_inc/objects/hm.h
user/libxm/include/comm.h
user/libxm/include/xm_inc/objects/commports.h
user/libxm/include/hm.h
user/libxm/include/hypervisor.h
user/libxm/include/arch/hypervisor.h
user/libxm/include/trace.h
user/libxm/include/xm_inc/objects/trace.h
user/libxm/include/status.h
user/libxm/include/xm_inc/objects/status.h
```

## 5.5 Partition reset

A partition reset is an unconditional jump to the partition entry point. The default entry point of a partition is obtained from the ELF partition image at the time of the partition creation.

There are two modes to reset a partition: `XM_WARM_RESET` and `XM_COLD_RESET`.

On a warm reset, the state of the partition is mostly preserved. Only the field `resetCounter` of the PCT is incremented, and the field `resetStatus` is set to the value given on the hypercall (see `XM_partition_reset()`).

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()` function when the partition is reset as result of this service.

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

## 5.6 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:** A hardware reset if forced. (See section 8.3.5).

*The set of actions done on a warm system reset are still under development.*

## 5.7 Scheduling

### 5.7.1 Slot identification

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

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.

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

### 5.7.2 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 plans. 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 5.13: core/include/objects/status.h

**`switchTime`:** The absolute time of the last plan switch request. After a reset (both warm and cold), the 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`).

## 5.8 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 5.14: Simple hypercall invocation.

Additionally to this low level hypercall, some function 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 a `stdio`.

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 5.15: 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.

## 5.9 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.

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* . A object descriptor is an integer, where the 16 least significant bits are a unique id of the port and the upper bits are reserved for internal use.

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

```c
#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 5.16: Partition_1

```c
#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 5.17: Partition_2

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

### 5.9.1 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.

## 5.10 Peripheral programming

The LEON2 processor implements a memory-mapped I/O for performing hardware input and output operations to the peripherals. There are two hypercalls to access I/O registers: XM_sparcv8_inport() and XM_sparcv8_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.

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);
StoreIoReg(port, ((oldValue&~(mask))|(ioMsg->msg[i]&mask)));
```

Listing 5.18: core/kernel/sparcv8/hypercalls.c

First off, the port is read, to get the value of the bits not allocated to the partitions, then the bits which 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 than the value previous written.

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

```
<Partition ..... >
    <HwResources>
        <IoPorts>
            <Restricted address="0x3000" 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.

## 5.11 Traps, interrupts and exceptions

### 5.11.1 Traps

A partition can not directly manage processor traps. XtratuM provides a para-virtualized trap system called *virtual traps*. XtratuM defines 256+32 traps. The first 256 traps correspond directly with to the hardware traps. The last 32 ones are defined by XtratuM.

The structure of the virtual trap table mimics the native trap table structure. Each entry is a 16 bytes large and contains the trap handler routine (which in the practice, is a branch or jump instruction to the real handler).

At boot time (or after a reset) a partition shall setup its virtual trap table and load the virtual `$tbr` register with the start address of the table. The `$tbr` register can be managed with the hypercalls: `XM_read_register32()` and `XM_write_register32()` with the appropriate register name:

```
#define TBR_REG32 0
```

If a trap is delivered to the partition and there is not a valid virtual trap table, then the health monitoring event XM_EM_EV_PARTITION_UNRECOVERABLE is generated.

### 5.11.2 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.

There are two groups of virtual interrupts:

**Hardware interrupts:** Correspond to the native hardware interrupts. Note that SPARC v8 defines only 15 interrupts (from 1 to 15), but XtratuM reserves 32 for compatibility with other architectures.

Interrupt 1 to 15 are assigned to traps 0x11 to 0x1F respectively (as in the native hardware).

**Extended interrupts:** Correspond to the XtratuM extended interrupts.

These interrupts are assigned from traps 0xE0 to 0xFF.

```
#define XM_VT_HW_FIRST          (0)
#define XM_VT_HW_LAST           (31)
#define XM_VT_HW_MAX            (32)

#ifdef CONFIG_LEON3FT

#define XM_VT_HW_INTERNAL_BUS_TRAP_NR (1+XM_VT_HW_FIRST)
#define XM_VT_HW_UART1_TRAP_NR    (2+XM_VT_HW_FIRST)
#define XM_VT_HW_IO_IRQ0_TRAP_NR  (3+XM_VT_HW_FIRST)
#define XM_VT_HW_IO_IRQ1_TRAP_NR  (4+XM_VT_HW_FIRST)
#define XM_VT_HW_TIMER_LATCH_TRAP_NR (7+XM_VT_HW_FIRST)
#define XM_VT_HW_TIMER1_TRAP_NR   (8+XM_VT_HW_FIRST)
#define XM_VT_HW_TIMER2_TRAP_NR   (9+XM_VT_HW_FIRST)
#define XM_VT_HW_TIMER3_TRAP_NR   (10+XM_VT_HW_FIRST)
#define XM_VT_HW_TIMER4_TRAP_NR   (11+XM_VT_HW_FIRST)
#define XM_VT_HW_PCI_TRAP_NR      (14+XM_VT_HW_FIRST)
#define XM_VT_HW_UART2_TRAP_NR    (17+XM_VT_HW_FIRST)
#define XM_VT_HW_UART3_TRAP_NR    (18+XM_VT_HW_FIRST)
#define XM_VT_HW_UART4_TRAP_NR    (19+XM_VT_HW_FIRST)
#define XM_VT_HW_UART5_TRAP_NR    (20+XM_VT_HW_FIRST)
#define XM_VT_HW_UART6_TRAP_NR    (21+XM_VT_HW_FIRST)

#else

#define XM_VT_HW_INTERNAL_BUS_TRAP_NR (1+XM_VT_HW_FIRST)
#define XM_VT_HW_UART2_TRAP_NR    (2+XM_VT_HW_FIRST)
#define XM_VT_HW_UART1_TRAP_NR    (3+XM_VT_HW_FIRST)
#define XM_VT_HW_IO_IRQ0_TRAP_NR  (4+XM_VT_HW_FIRST)
#define XM_VT_HW_IO_IRQ1_TRAP_NR  (5+XM_VT_HW_FIRST)
#define XM_VT_HW_IO_IRQ2_TRAP_NR  (6+XM_VT_HW_FIRST)
#define XM_VT_HW_IO_IRQ3_TRAP_NR  (7+XM_VT_HW_FIRST)
#define XM_VT_HW_TIMER1_TRAP_NR   (8+XM_VT_HW_FIRST)
#define XM_VT_HW_TIMER2_TRAP_NR   (9+XM_VT_HW_FIRST)
#define XM_VT_HW_DSU_TRAP_NR      (11+XM_VT_HW_FIRST)
#define XM_VT_HW_PCI_TRAP_NR      (14+XM_VT_HW_FIRST)
```

```
#endif

#define XM_VT_EXT_FIRST        (0)
#define XM_VT_EXT_LAST         (31)
#define XM_VT_EXT_MAX          (32)

#define XM_VT_EXT_HW_TIMER      (0+XM_VT_EXT_FIRST)
#define XM_VT_EXT_EXEC_TIMER    (1+XM_VT_EXT_FIRST)
/*                              (2+XM_VT_EXT_FIRST) not used*/
#define XM_VT_EXT_SHUTDOWN      (3+XM_VT_EXT_FIRST)
#define XM_VT_EXT_SAMPLING_PORT (4+XM_VT_EXT_FIRST)
#define XM_VT_EXT_QUEUING_PORT  (5+XM_VT_EXT_FIRST)

#define XM_VT_EXT_CYCLIC_SLOT_START (8+XM_VT_EXT_FIRST)

#define XM_VT_EXT_MEM_PROTECT   (16+XM_VT_EXT_FIRST)

/* Inter-Partition Virtual Interrupts */
#define XM_MAX_IPVI            CONFIG_MAX_NO_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 5.19: core/include/guest.h

Both, extended and hardware interrupts can be routed to a different interrupt vector through the `XM_route_irq()` hypercall, this hypercall enables a partition to select the most suitable vector to be raised.

All hardware and extended interrupts can be masked through the following hypercalls: `XM_clear_irqmask()` and `XM_set_irqmask()`. Besides, all these set of interrupts can be globally disabled/enable by using the `XM_sparc_get_psr()` and `XM_sparc_set_psr()` hypercalls, mimicking the way the underneath architecture works.

### 5.11.3 Exceptions

Exceptions are the traps triggered by the processor in response to an internal condition. Some exceptions are caused by normal operation of the processor (e.g. register window over/underflow) but others are caused by abnormal situations (e.g. invalid instruction).

Error related exception traps, are managed by XtratuM thorough the health monitoring system.

```
#define DATA_STORE_ERROR 0x2b // 0
#define INSTRUCTION_ACCESS_MMU_MISS 0x3c // 1
#define INSTRUCTION_ACCESS_ERROR 0x21 // 2
#define R_REGISTER_ACCESS_ERROR 0x20 // 3
#define INSTRUCTION_ACCESS_EXCEPTION 0x1 // 4
#define PRIVILEGED_INSTRUCTION 0x03 // 5
#define ILLEGAL_INSTRUCTION 0x2 // 6
#define FP_DISABLED 0x4 // 7
#define CP_DISABLED 0x24 // 8
#define UNIMPLEMENTED_FLUSH 0x25 // 9
#define WATCHPOINT_DETECTED 0xb // 10
```

```
//#define WINDOW_OVERFLOW 0x5
//#define WINDOW_UNDERFLOW 0x6
#define MEM_ADDRESS_NOT_ALIGNED 0x7 // 11
#define FP_EXCEPTION 0x8 // 12
#define CP_EXCEPTION 0x28 // 13
#define DATA_ACCESS_ERROR 0x29 // 14
#define DATA_ACCESS_MMU_MISS 0x2c // 15
#define DATA_ACCESS_EXCEPTION 0x9 // 16
#define TAG_OVERFLOW 0xa // 17
#define DIVISION_BY_ZERO 0x2a // 18
```

Listing 5.20: core/include/sparcv8/irqs.h

If the health monitoring action associated with the HM event is XM_HM_AC_PROPAGATE, then the same trap number is propagated to the partition as a virtual trap. The partition code is then in charge of handling the error.

## 5.12 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)
```

Listing 5.21: 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. These extended interrupts correspond with traps (256+XM_VT_EXT_HW_TIMER) and (256+XM_VT_EXT_EXEC_TIMER) respectively.

### 5.12.1 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$\mu$sec. Its precision is not as accurate as that of the XM_HW_CLOCK due to the errors introduced by the partition switch.

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);
}
```

## 5.13 Watchdog

XtratuM provides one software watchdog, called XM watchdog. It facilitates a mechanism to recover the system when hang. The XM watchdog requires the underlying hardware watchdog enabled to fully work. The system partitions can load the XM watchdog with an interval of time using the `XM_reload_watchdog()`. The XM watchdog reloads the hardware watchdog in each partition context switch, if the first one has not expired. The XM watchdog mechanism may be used without hardware watchdog. In this case, the XM watchdog does not perform any action even when the XM watchdog is active. When the XM watchdog is expired, the hardware watchdog is not reloaded.

The XM watchdog is globally defined for all partitions. That helps the system developers to abstract the use of the hardware watchdog. The XM watchdog can also be used without configuring the hardware watchdog, allowing developers debug the system in simulators or development boards without modifying the partition code.

XtratuM provides the `XM_reload_watchdog()` hypercall to reload the XM watchdog. This hypercall receives a budget in microseconds as parameter. The XM watchdog is armed with that budget. When the counter reachs 0, the watchdog expires. Only system partitions can reload the XM watchdog for safety reasons.

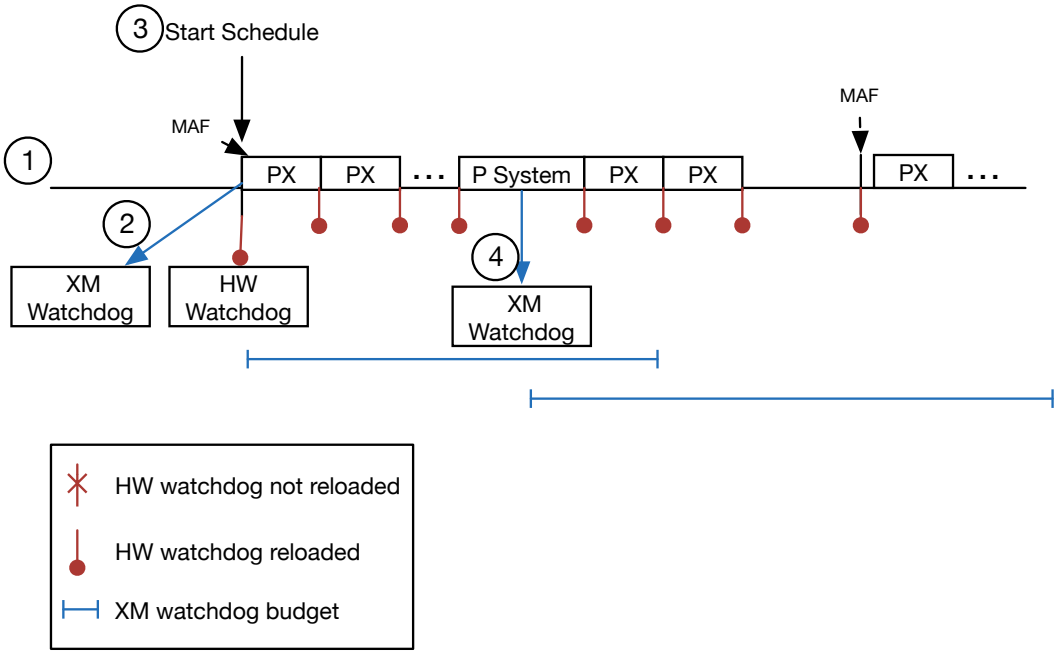



Figure 5.1: Scenario 1. XM Watchdog.

Figure 5.1 shows a scenerio where the XM watchdog does not expires: At system start-up (1), the XM watchdog is initialised short before the first context switch is produced (2). When the first context switch is produced, the HW watchdog is reloaded by XtratuM by first time if the XM watchdog has not

expired (3). After that, the HW watchdog is reloaded in each partition context switch as long as the XM watchdog has not expired. When the XM watchdog is reloaded by a system partition (4), he XM watchdog interval is updated.

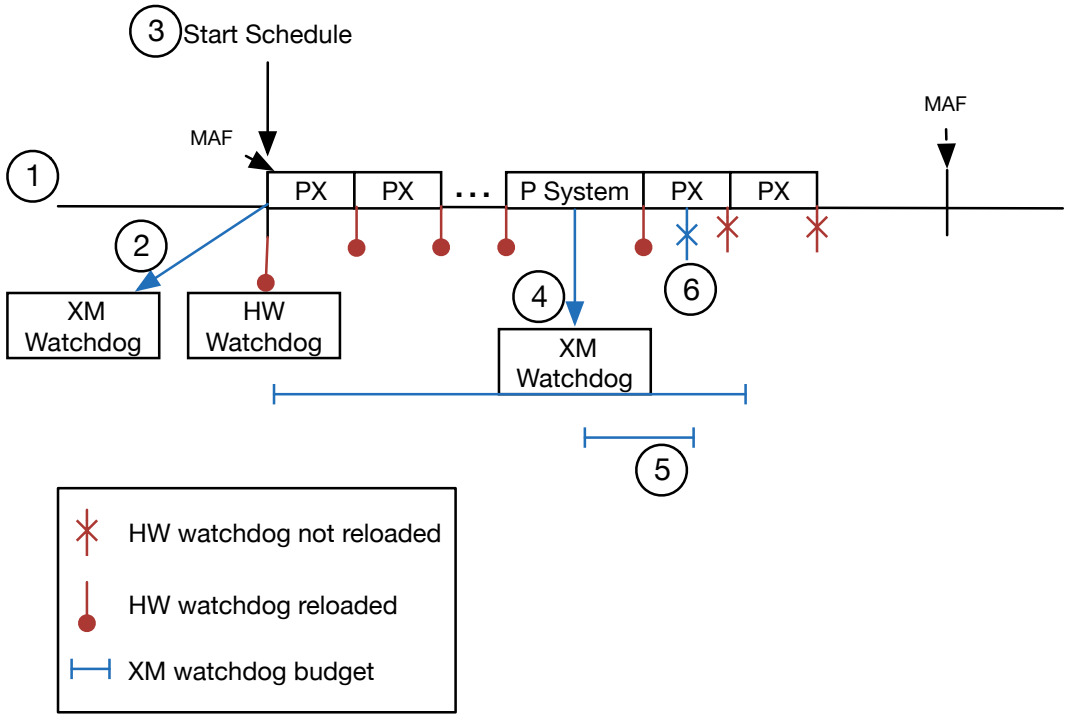



Figure 5.2: Scenario 2. XM Watchdog expires.

Figure 5.2 shows a scenario where the XM watchdog expires: At system start-up (1), the XM watchdog is initialised short before the first context switch is produced (2). When the first context switch is produced, the HW watchdog is reloaded by XtratuM by first time if the XM watchdog has not expired (3). After that, the HW watchdog is reloaded in each partition context switch as long as the XM watchdog has not expired. When the XM watchdog is reloaded by a system partition (4), he XM watchdog interval is updated. But in this case the new interval is shorter (5). After the XM watchdog expires (6), the HW watchog is not reloaded.

It is neccesary to configure the XM watchdog in the system configuration file (XMCF). In this configuration is specified the interval value to be reloaded in the hardware watchdog and the initialisation interval for the XM watchdog. See subsection 5.13.1.

The hardware watchdog devices are platform dependant. The device used is configured using the XtratuM source code configuration menu (section 8.1). The GR712RC board supports one hardware watchdog, see subsection 5.13.2.

NOTE: The XM watchdog can be used even if no hardware watchdog is configured. In that case, no action is performed by the XM watchdog.

### 5.13.1 Configuration

The XtratuM watchdog has to be properly configured in the XM_CF configuration file. An example of watchdog configuration in the XM_CF configuration file can be shown in Listing 5.22.

```
<SystemDescription xmlns="http://www.xtratum.org/xm-3.x" version="1.0.0"
name="timers"> ...
```

```
<XMHypervisor console="Uart" >
  <PhysicalMemoryArea size="512KB" />
  <Watchdog hwReload="100ms" initialValue="20ms" />
</XMHypervisor>
...
```

Listing 5.22: Watchdog configuration example

There are two required attributes:

- `hwReload` Reload value for hardware watchdog.
- `initialValue` Initial reload value of the XtratuM watchdog.

Note that if the XM watchdog is not configured, the `hwReload` attribute is supposed to be 0. So, in case an application actives the XM watchdog during runtime, the value loaded in the hardware watchdog will be 0.

### 5.13.2 GR712 Hardware watchdog driver

The GR712RC board includes a general pourpose timer unit, it is made up of four different timers. The last timer can act as watchdog, driving the watchdog output signal WDOGN when expired (See [GR712RC User Manual, section 11]).

This device is supported by XtratuM to be used as hardware watchdog timer. The device can be activated in the configuration menu of XtratuM.

When the GR712 hardware watchdog is used, XtratuM is responsible for managing it. The first time XtratuM scheduler is activated, the hardware watchdog is reloaded , with the configured value, if the XtratuM watchdog has not expired. That is the normal behaviour of the XtratuM watchdog mechanism. This reloading process is done after XtratuM and the partitions are loaded in memory, so the system integrator has to guarantee that the hardware watchdog will not expire before XtratuM takes over it.

NOTE: After a reset, the GR712 GP timer is loaded with an initial prescaler value [GR712RC User Manual, section 11.3] that provides an interval of 50seconds, at 80Mhz, before the watchdog expires. This time is long enough to XtratuM takes over the hardware watchdog. However, the GRMON tools, or the bootloader, could change the prescaler value, decrementing this interval duration.

NOTE: The system integrator shall guarantee the hardware watchdog is deactivated, or the initial budget is long enough to avoid a system reset before XtratuM starts managing it. Note that XtratuM sets the prescaler tick interval to one microsecond ($1usec$) when it initialises the clocks.

## 5.14 Processor management

Currently only the `$tbr` processor control register has been virtualised. This register should be loaded (with the hypercall `XM_write_register32()`) with the address of the partition trap table. This operation is usually done only once when the partition boots (see listing **??**).

### 5.14.1 Managing stack context

The SPARC v8 architecture (following the RISC ideas) tries to simplify the complexity of the processor by moving complex management tasks to the compiler or the operating system. One of the most particular features of SPARC v8 is the register window concept, and how it should be managed.

Both, register window overflow and underflow cause a trap. This trap has to be managed in supervisor mode and with traps disabled (bit ET in the `$psr` register is unset) to avoid overwriting valid registers. It is not possible to emulate efficiently this behaviour.

XtratuM provides a transparent management of the stack. Stack overflow and underflow is directly managed by XtratuM without the intervention of the partition. The partition code shall load the stack register with valid values.

In order to implement a content switch inside a partition (if the partition is a multi-thread environment), the function `XM_sparcv8_flush_regwin()` can be used to flush (spill) all register windows in the current CPU stack. After calling this function, all the register windows, but the current one, are stored in RAM and then marked as free. The partition context switch code should basically carry out the next actions:

1. call `XM_sparcv8_flush_regwin()`
2. store the current "g", "i" and "l" registers in the stack
3. switch to the new thread's stack and
4. restore the same set of registers.

Note that there is no complementary function to reload (fill) the registers, because it is done automatically.

The `XM_sparcv8_flush_regwin()` service can also be used set the processor in a know state before executing a block of code. All the register windows will be clean and no window overflow will happen during the next 7 nested function calls.

## 5.15 Tracing

### 5.15.1 Trace messages

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:

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

typedef struct xmTraceEvent xmTraceEvent_t;
```

Listing 5.23: core/include/objects/trace.h

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

**`payload`:** Used defined data.

### 5.15.2 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**

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]

## 5.16 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 irqLastOccurence[CONFIG_NO_HWIRQS];
} xmPartitionStatus_t;
```

Listing 5.24: 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 interation. */
    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 irqLastOccurence[CONFIG_NO_HWIRQS];
#ifdef CONFIG_LEON_EDAC_SUPPORT
    xm_u32_t edacErrorCounter;
```

```
#endif
} xmSystemStatus_t;
```

Listing 5.25: 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.

## 5.17 Memory management

XtratuM implements a flat memory space on the SPARC v8 architecture (LEON2 and LEON3 processors). 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.

Two different hardware features can be used to implement memory protection:

**Write Protection Registers (WPR):** In the case that there is no MMU support, then it is possible to use the WPR device of the LEON2 and LEON3 processors. The WPR device can be programmed to raise a trap when the processor tries to write on a configured address range.

Since read memory operations are not controlled by the WPR, it is not possible to enforce complete (read/write) memory isolation in this case. Also, due to the internal operation of the WPR device, all the memory allocated to each partition has to be contiguous and has to meet the following conditions:

- The size shall be greater than or equal to 32KB.
- The size shall be a power or two.
- The start address shall be a multiple of the size.

**Memory Management Unit (MMU):** If the processor has MMU, and XtratuM has been compiled to use it, then XtratuM will map the memory areas allocated to each partition using the page size that better fits the memory areas sizes. To minimize the number of page table reserved and used by XtratuM it is recommended that memory areas be multiple of (4KB, 256KB or 16MB) and be aligned to it. For example:

- A memory area of 252KB aligned to 4KB will need a total of 63 page table of level 3 (63KB)
- A memory area of 256KB aligned to 4KB but not aligned to 256KB will need a total of 64 page table of level 3 (64KB)
- A memory area of 256KB aligned to 256KB will need a total of 1 page table of level 2 (256bytes)

XtratuM always reserves and uses the minimum number of page table needed to map a memory area. The aligment supported by the processor and the size of each page table is described in SPARCv8 User’s Manual.

The MMU is used only as a MPU (memory protection unit), i.e, the virtual and physical addresses are the same. Only the protections bits of the pages are used. As a result, each partition shall be compiled and linked to the designated addresses where they will be loaded and executed.

The memory protection mechanism employed is a source code configuration option. See section 8.1.

The memory areas allocated to a partition are defined in the XM_CF file. The executable image shall be linked to be executed in those allocated memory areas.

### Configuration

A partition can statically allocates 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, booter and other partitions.

An example of the memory areas definition in the XM_CF configuration file can be shown in 5.26.

```
<MemoryLayout>
  <Region type="stram" start="0x40000000" size="4MB"/>
  <Region type="sdram" start="0x60000000" size="8KB"/>
  <Region type="sdram" start="0xfffff000" size="4KB"/>
</MemoryLayout>
...

<XMHypervisor console="Uart" >
  <PhysicalMemoryArea size="512KB" />
</XMHypervisor>
...
<ResidentSw>
  <PhysicalMemoryAreas>
    <Area start="0x60300000" size="512KB"/>
  </PhysicalMemoryAreas>
</ResidentSw>
...
<PartitionTable>
  <Partition id="0" name="Partition0" console="Uart">
    <PhysicalMemoryAreas>
      <Area start="0x40100000" size="256KB" />
      <Area start="0x60200000" size="64KB" mappedAt="0x10000" />
      <Area start="0x60210000" size="64KB" mappedAt="0x20000" />
      <Area start="0x60220000" size="64KB" flags= "uncacheable" />
      <Area start="0xfffff000" size="4KB" />
    </PhysicalMemoryAreas>
    <TemporalRequirements duration="500ms" period="500ms"/>
  </Partition>
  <Partition id="1" name="Partition1" flags="system" console="Uart">
    <PhysicalMemoryAreas>
      <Area start="0x40140000" size="256KB" mappedAt="0x40000000" />
      <Area start="0x60210000" size="64KB" flags= "read-only" />
      </PhysicalMemoryAreas>
    <TemporalRequirements duration="500ms" period="500ms"/>
  </Partition>
</PartitionTable>
..
```

Listing 5.26: Memory areas allocation example

The memory layout defined for the board is:

The memory areas defined and the owners is detailed in the next table:

| Memory type | Initial | End addres | Size |
|---|---|---|---|
| STRAM | 40000000 | 403FFFFF | 4 MB |
| SDRAM | 60000000 | 607FFFFF | 8 MB |
| STRAM | FFFFF000 | FFFFFFFF | 4 KB |

| Component | Initial | End addres | Size | Attributes | Mapped at |
|---|---|---|---|---|---|
| XtratuM | 40000000 | 400FFFFF | 1 MB | | |
| Partition0 | 40100000 | 4013FFFF | 256 KB | | |
| Partition1 | 40140000 | 4017FFFF | 256 KB | | |
| Partition0 | 60200000 | 6020FFFF | 64 KB | | 10000 |
| Partition0 | 60210000 | 6021FFFF | 64 KB | rw | 20000 |
| Partition1 | 60210000 | 6021FFFF | 64 KB | ro | |
| Partition0 | 60220000 | 6022FFFF | 64 KB | Uncacheable | |
| RSW | 60300000 | 6037FFFF | 512 KB | | |
| Container | 60400000 | 6047FFFF | 512 KB | | |
| Partition0 | FFFFF000 | FFFFFFFF | 4 KB | | |

NOTE: The container address is set in the XtratuM configuration process. In order to change it, a new XtratuM compilation and installations has to be performed.

In this example, Partition0 defines the following memory areas:

- A memory area of 256Kb that is allocated to the physical memory ”0x40100000”.
- A memory area of 64Kb allocated in the physical memory at ”0x60200000” but mapped at ”0x1000”. So, internaly the partition can access to this area using this virtual address.
- A memory area of 64Kb allocated in the physical memory at ”0x60210000”. This partition can read and write on this area.
- A memory area of 64Kb allocated in the physical memory at ”0x60220000” that is not cacheable.
- A memory area of 4Kb allocated in the physical memory at ”0xfffff000” but mapped at ”0x2000”.

On the other hand, Partition1 defines:

- A memory area of 256Kb that is allocated to the physical memory ”0x40140000” and mapped at ”0x40000000”.
- A memory area of 64Kb allocated in the physical memory at ”0x60210000”. This partition can only read on this area.

### 5.17.1 EDAC support

For the LEON3FT processor and the GR712RC board, XtratuM optionally supports EDAC in the LEON3FT system’s memory controller. When XtratuM is compiled with EDAC support enabled, the hypervisor manages EDAC errors as described in the listing 5.27: correctable EDAC errors are solved and tracked by incrementing an internal EDAC error counter. On the other hand, uncorrectable errors cause to raise XM_HM_EV_SPARCV8_UNCORRECTABLE_EDAC_ERROR HM event, storing the failing address as part of the event.

```
if an EDAC error has been detected then
   if the EDAC error is correctable then
      Increment XM internal EDAC error counter;
```

```
        -- Read-only memory areas cannot be scrubbed.
        if EDAC failing address is within a read-write memory area then
            Scrub the content of the failing address;
        end if;
        Re-enable EDAC mechanism;
    else -- uncorrectable EDAC error
        Re-enable EDAC mechanism;
        -- The XM_HM_EV_SPARCV8_UNCORRECTABLE_EDAC_ERROR HM event is raised,
        -- the failing address is tracked as the payload of this HM event.
        Raise XM_HM_EV_SPARCV8_UNCORRECTABLE_EDAC_ERROR HM event;
    end if;
end if;
```

Listing 5.27: EDAC handling algorithm

Note that EDAC mechanism must have been enabled in the board before loading the software. For instance, *GRMON* provides `-edac` option enabling to start the system with EDAC already set.

Note that the following an uncorrectable EDAC error in the following situations can left the system in an undefined or inconsistent state:

- There is an uncorrectable EDAC error in, at least, one instruction or data at the start of a trap when XtratuM runs with `PSR.ET` bit disabled. This scenario causes the processor to enter in error mode.
- There is an uncorrectable EDAC error in, at least, one instruction or data just before returning from a trap. XtratuM must disable `PSR.ET` bit just before executing the `rett` instruction. This scenario causes the processor to enter in error mode.
- There is an uncorrectable EDAC error in, at least, one instruction or data in the execution path for managing a `XM_HM_EV_SPARCV8_UNCORRECTABLE_EDAC_ERROR` event. This scenario may cause and endless loop.

The first two situations cannot be prevented by the software. The last one may be mitigated but cannot discarded.

Note that the EDAC mechanism provided by the GR712RC (without considering Reed-Solomon CRC) only is able to fix one bit and detect two. Changes in more bits can potentially be detected as correctable EDAC errors, where the corrected value provided by the GR712RC mechanism is wrong.

The EDAC error counter is returned to system partitions as the `edacErrorCounter` field of the `xmSystemStatus_t` through the `XM_get_system_status()` hypercall.

## 5.18 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 to XtratuM about its idle state (see `XM_idle_self()`). In the current implementation, XtratuM does nothing while a partition is idle, that is, other partition is not executed; but it opens the possibility to use this wasted time in internal bookkeeping or other maintenance activities. Also, energy saver actions can be done during this idle time.

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

## 5.19 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 6.4.1) 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 structure which defines 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.

These 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 shall 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.

## 5.20 Assembly programming

This section describes the assembly programming convention, in order to invoke the XtratuM hypercalls.

The register assignment convention for calling a hypercall is:

`%o0` Holds the hypercall number.

`%o1 - %o5` Holds the parameters to the hypercall.

Once the processor registers have been loaded, a **`ta`** instruction to the appropriate software trap number shall be called, see section 6.2.

The return value is stored in register **`%o0`**.

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

```
mov  0xa , %o0 ; __GET_TIME_NR
mov  %i0, %o1
mov  %i1, %o2
ta   0xf0
cmp  %o0, 0     ; XM_OK == 0
bne   <error>
```

In SPARC v8, the `get_time_nr` constant has the value "`0xa`"; "`%i0`" holds the clock id; and "`%i1`" is a pointer which points to a `xmTime_t` variable. The return value of the hypercall is stored in "`%o0`" and then checked if `XM_OK`.

Below is the list of normal hypercall number constants (listing **??**) and assembly hypercalls (listing **??**):

```
#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 sparc_inport_nr 22
#define sparc_outport_nr 23
#define sparc_write_tbr_nr 24

#define sparc_disable_sdp_nr 25
#define sparc_outport_sdp_nr 26
#define sparc_outport_msg_nr 27
#define reserved_0 28
#define reserved_1 39

#define __RESET_VCPU_NR 30
#define __HALT_VCPU_NR 31
#define __GET_VCPUID_NR 32
#define __RAISE_IPVI_NR 33
```

Listing 5.28: core/include/sparcv8/hypercalls.h

```
#define sparc_iret_nr 0
#define sparc_flush_regwin_nr 1
#define sparc_get_psr_nr 2
#define sparc_set_psr_nr 3
#define sparc_set_pil_nr 4
#define sparc_clear_pil_nr 5
#define sparc_ctrl_winflow_nr 6
```

Listing 5.29: core/include/sparcv8/hypercalls.h

The file “core/include/sparckv8/hypercalls.h” has additional services for the SPARC v8 architecture.

### 5.20.1 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_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);
```

Listing 5.30: 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.

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

```
xm_s32_t XM_hm_status(xmHmStatus_t *hmStatusPtr) {

    if (!hmStatusPtr) {
        return XM_INVALID_PARAM;
    }
    return XM_ctrl_object(OBJDESC_BUILD(OBJ_CLASS_HM, XM_HYPERVISOR_ID, 0),
        XM_HM_GET_STATUS, hmStatusPtr);
}
```

Listing 5.31: user/libxm/common/hm.c

## 5.21 Manpages summary

Below is a summary of the manpages. A detailed information is provided in the document "*Volume 4: Reference Manual*".

| **System Management** | |
|---|---|
| XM_get_system_status | Get the current status of the system. |
| XM_halt_system | Stop the system. |
| XM_reset_system | Reset the system. |
| **Partition Management** | |
| XM_get_partition_mmap | This function returns a pointer to the memory map table (MMT). |
| XM_get_partition_status | Get the current status of a partition. |
| XM_halt_partition | Terminates a partition. |
| XM_idle_self | Idles the execution of the calling partition. |
| XM_params_get_PCT | Return the address of the PCT. |
| XM_reset_partition | Reset a partition. |
| XM_resume_partition | Resume the execution of a partition. |
| XM_shutdown_partition | Send a shutdown interrupt to a partition. |
| XM_suspend_partition | Suspend the execution of a partition. |

| | |
|---|---|
| **Time Management** | |
| XM_get_time | Retrieve the time of the clock specified in the parameter. |
| XM_set_timer | Arm a timer. |
| **Plan Management** | |
| XM_get_plan_status | Return information about the scheduling plans. |
| XM_set_plan | Request a plan switch at the end of the current MAF. |
| **Inter-Partition Communication** | |
| XM_create_queuing_port | Create a queuing port. |
| XM_create_sampling_port | Create a sampling port. |
| XM_get_queuing_port_info | Get the info of a queuing port. |
| XM_get_queuing_port_status | Get the status of a queuing port. |
| XM_get_sampling_port_info | Get the info of a sampling port. |
| XM_get_sampling_port_status | Get the status of a sampling port. |
| XM_memory_copy | Copy copies data from/to address spaces. |
| XM_read_sampling_message | Reads a message from the specified sampling port. |
| XM_receive_queuing_message | Receive a message from the specified queuing port. |
| XM_send_queuing_message | Send a message in the specified queuing port. |
| XM_write_sampling_message | Writes a message in the specified sampling port. |
| **Health Monitor Management** | |
| XM_hm_raise_event | The invoking partition generates a new hm event. |
| XM_hm_read | Read one or more health monitoring log entries. |
| XM_hm_status | Get the status of the health monitoring log stream. |
| **Trace Management** | |
| XM_trace_event | Records a trace entry. |
| XM_trace_read | Read a trace event. |
| XM_trace_status | Get the status of a trace stream. |
| **Interrupt Management** | |
| XM_clear_irqmask | Unmask interrupts. |
| XM_clear_irqpend | Clear pending interrupts. |
| XM_disable_irqs | Disable interrupts. |
| XM_enable_irqs | Enables interrupts. |
| XM_route_irq | Link an interrupt with the vector generated when the |
| XM_set_irqmask | Mask interrupts. |
| XM_set_irqpend | Set some interrupts as pending. |
| **Miscelaneous** | |
| XM_get_git_by_name | Returns the identifier of an entity defined in the configuration file. |
| XM_write_console | Print a string in the hypervisor console. |
| **Sparcv8 specific** | |
| XM_sparc_clear_pil | Clear the PIL field of the PSR (enable interrupts). |
| XM_sparc_ctrl_winflow | Check manually the state of register windows. |
| XM_sparc_disable_sdp | Disable the SDP to enable single-byte writing in EEPROM. |
| XM_sparc_flush_regwim | Save the contents of the register window. |
| XM_sparc_get_psr | Get the ICC and PIL flags from the virtual PSR processor register. |

| | |
|---|---|
| XM_sparc_inport | Read from a hardware I/O port. |
| XM_sparc_outport | Write in a hardware I/O port. |
| XM_sparc_outport_sdp | Write in EEPROM atomically or enable SDP mode. |
| XM_sparc_set_pil | Set the PIL field of the PSR (disallow interrupts). |
| XM_sparc_set_psr | Set the ICC and PIL flags on the virtual PSR processor register. |

This page is intentionally left blank.

# Chapter 6

# Binary Interfaces

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

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. These files may be created in a little-endian system (like the PC) while LEON2 is a big-endian one.

XtratuM conforms the following conventions:

| Unsigned | Signed | Size (bytes) | Alignment (bytes) |
|---|---|---|---|
| xm_u8_t | xm_s8_t | 1 | 1 |
| xm_u16_t | xm_s16_t | 2 | 4 |
| xm_u32_t | xm_s32_t | 4 | 4 |
| xm_u64_t | xm_s64_t | 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:

```
// 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;
```

Listing 6.1: core/include/sparcv8/arch_types.h

```
// Extended types
typedef long xmLong_t;
typedef xm_u32_t xmWord_t;
#define XM_LOG2_WORD_SZ 5
typedef xm_s64_t xmTime_t;
#define MAX_XMTIME 0x7fffffffffffffLL
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/sparcv8/arch_types.h

For future compatiblity, 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:

```
#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

A hypercall is implemented by a trap processor instruction that transfers the control to XtratuM code, and sets the processor in supervisor mode.

There are two kind of hypercalls: normal and assembly. Each type of hypercall use a different trap number:

```
#define XM_HYPERCALL_TRAP 0xF0
#define XM_ASMHYPERCALL_TRAP 0xF1
```

Listing 6.4: core/include/sparcv8/xm_def.h

The `XM_ASMHYPERCALL_TRAP` hypercall entry is needed for the `XM_sparcv8_flush_regwin()`, `XM_sparcv8_iret()` and `XM_sparcv8_get_flags()` calls. In this case, the XtratuM entry code does not prepare the processor to execute "C" code.

## 6.3 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.

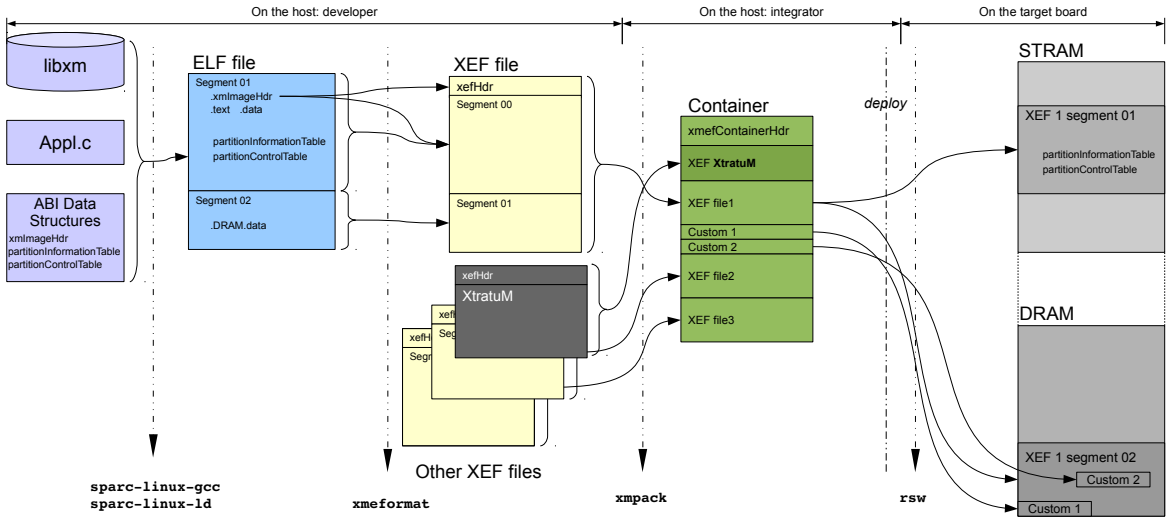



Figure 6.1: Executable formats.

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. Which allows to detect transmission errors.
- Compress the content. This feature greatly reduce the space of the image; consequently the deploy time.
- Encrypt the content. 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.

## 6.4 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 is 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.

Note: all the addresses of partition image are absolute addresses which refer to the target RAM memory locations.

### 6.4.1 Partition image header

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

```
struct xmImageHdr {
#define XMEF_PARTITION_MAGIC 0x24584d69 // $XMi
    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;
```

Listing 6.5: core/include/xmef.h

**sSignature and eSignature:** Holds the **s**tart and **e**nd signatures which identifies 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.

The current values of these fields are:

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

#define XM_API_VERSION 1
#define XM_API_SUBVERSION 0
#define XM_API_REVISION 0
```

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 5.19.

**customFileTab:** Table information about the customisation files.

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

Listing 6.7: core/include/xmef.h

**sAddr:** Address where the customisation file shall be loaded.

**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 5.4.

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.

### 6.4.2 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.

```
typedef struct {
    xm_u32_t magic;
    xm_u32_t xmVersion; // XM version
    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;
    xmId_t id;
    // Copy of kthread->ctrl.flags
    xm_u32_t flags;
#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)
#define PARTITION_DCACHE_ENABLED_F (1<<7)
#define PARTITION_ICACHE_ENABLED_F (1<<8)
    xm_u32_t imgStart;
    xm_u32_t hwIrqs; // Hw interrupts belonging to the partition
    xm_s32_t noPhysicalMemAreas;
    xm_s32_t noCommPorts;
    xm_u8_t name[CONFIG_ID_STRING_LENGTH];
    xm_u32_t iFlags; // As defined by the ARCH (ET+PIL in sparc)
    xm_u32_t hwIrqsPend; // pending hw irqs
    xm_u32_t hwIrqsMask; // masked hw irqs

    xm_u32_t extIrqsPend; // pending extended irqs
    xm_u32_t extIrqsMask; // masked extended irqs
    struct pctArch arch;
    struct {
        xm_u32_t noSlot:16, releasePoint:1, reserved:15;
        xm_u32_t id;
```

```
        xm_u32_t slotDuration;
    } schedInfo;
    xm_u16_t trap2Vector[NO_TRAPS];
    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;
#ifdef CONFIG_MMU
#define _ARCH_PTDL1_REG ptdL1
    volatile xm_u32_t faultStatusReg;
    volatile xm_u32_t faultAddressReg;
#endif
};
```

Listing 6.9: core/include/sparcv8/guest.h

**signature:** Signature to identity this data structure as a PIT.

**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.

**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.

**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.

**name:** Name of the partition.

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

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

**hwIrqsMask:** Bitmap of the extended interrupts allocated to the partition delivered to the partition.

**extIrqsMask:**

In the current version there is no specific architecture data.

## 6.5 XEF format

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

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

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

The tool `xmeformat` converts from ELF or plain data files to XEF format, see chapter 10.

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

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

#define XEF_ARCH_SPARCv8 0x400
#define XEF_ARCH_MASK 0xff00
    xm_u32_t flags;
    xm_u8_t digest[XM_DIGEST_BYTES];
    xm_u8_t payload[XM_PAYLOAD_BYTES];
    xmSize_t fileSize;
    xmAddress_t segmentTabOffset;
    xm_s32_t noSegments;
    xmAddress_t customFileTabOffset;
    xm_s32_t noCustomFiles;
    xmAddress_t imageOffset;
    xmSize_t imageLength;
    xmSize_t deflatedImageLength;
    xmAddress_t pageTable;
    xmSize_t pageTableSize;
    xmAddress_t xmImageHdr;
    xmAddress_t entryPoint;
} __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 all 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 the security level[1]. Note that the digest field is used to detect not deliberate modifications rather than intentional attacks. In this scenario, the MD5 is a good choice.

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

The content of this field is used neither by XtratuM nor the resident software.

**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.

**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.

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

**entryPoint:** Address of the starting function.

Additionally, analogically 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 **??**) which describes each one of the sections of the image (custom data XEF files have only one section).

```
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

[1] According to our tests, the time spent by more sophisticated digest algorithms such as SHA-2, Tiger or Whirlpool in the LEON3 processor was not acceptable. As illustration, 100 Kbytes took several seconds to be digested by a SHA-2 algorithm in this processor.

**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 from the memory required to be executed (for example a BSS usually requires more memory once loaded into memory).

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

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

### 6.5.1 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.

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.

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.

| 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.

## 6.6 Container format

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

The tool `xmpack` manages container files, see chapter 10.

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 chage 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 he 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 the file metadata (name of the files) and the content 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 remainder sections of the file.
2. The component table section, which contains an array of `xmefComponent` structures. Each element contains information of 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 1
#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.

**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.

**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.

**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 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. Which contains the following fields:

```
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.

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

**noCustomFiles:** Number of customisation files of this component, including.

**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 10.3.1 tool.

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

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.

# Chapter 7

# Booting

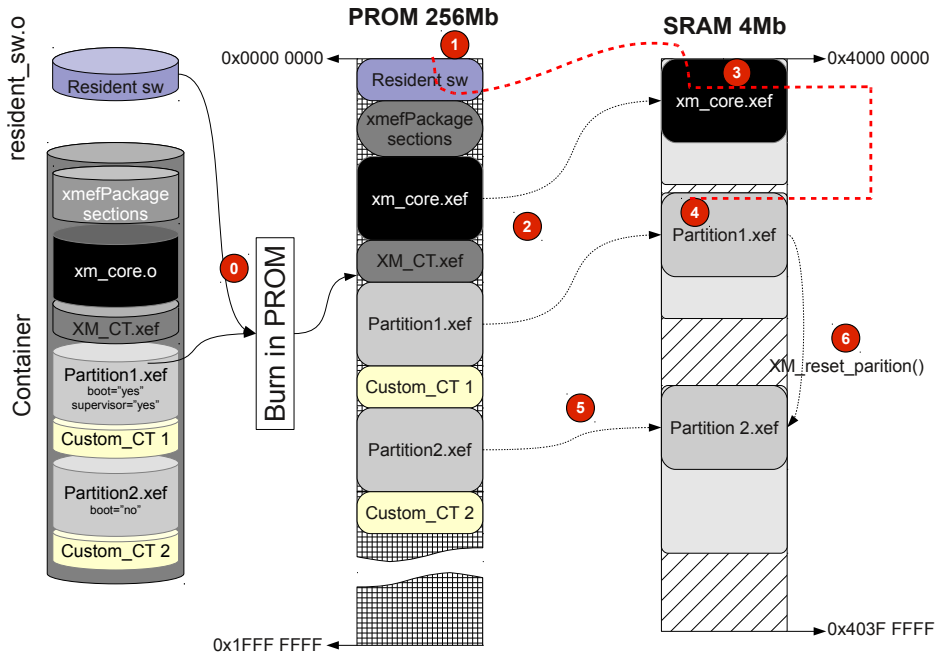



Figure 7.1: Booting sequence.

In the standard boot procedure of the LEON2 processor, the program counter register is initialized with the address `0x00000000`. Contrarily to other computers, the PROM of the board does not have any kind of resident software-like booter[1] that takes the control of the processor after the reset.

We have developed a small booting code called *resident software*, which is in charge of the initial steps of booting the computer. This software is not part of the container produced by the `xmpack` tool. It is prepended to the container by the burning script.

The board has two kind of RAM memory: SRAM (4Mb) and SDRAM (128Mb).

[1] Known as BIOS in the personal computer area.

## 7.1 Boot configuration

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.

After starting, XtratuM assumes that the partitions configured 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 sequence is sketched in figure 7.1.

⓪ The deployment tool to burn the PROM of the board writes first the resident software and right after it the container, which should contain the XtratuM core and the booting partitions components. Note that the container can also contain the non-booting partitions.

① When the processor is started (reset) the resident software is executed. It is a small code that performs the following actions:

1. Initializes a stack (required to execute "C" code).
2. Installs a trap handler table (only for the case that its code would generate a fault, XtratuM installs a new trap handler during its initialisation).
3. Checks that the data following the resident software in the PROM is a container (a valid signature), and seeks the XtratuM hypervisor through container (a valid signature).
4. Copies the XtratuM hypervisor and booting partitions into RAM memory (②).
5. Jumps to the entry point of the hypervisor in RAM memory.

③ XtratuM assumes that interrupts are disabled. So, the first code has to be written in assembly (`code/kernel/sparcv8/head.s`), and performs the next actions: general purpose registers are cleared; memory access rights are cleared; PSR[2], WIM[3], TBR[4] and Y[5] processor control registers are initialized; sets up a working stack; and jumps to "C" code (`code/kernel/setup.c`).

The `setup()` function carries out the following actions:

1. Checks the existence and the correctness of the xm_cf structure. In case that this structure is not found, the system is halted. This halt operation consists in either entering in an endless loop or setting the processor in power down mode.
2. Initializes the internal console.
3. Initializes the interrupt controller.
4. Detects the processor frequency (information extracted from the XML configuration file).
5. Initializes memory manager (enabling XtratuM to keep track of the use of the physical memory).

[2]PSR: Processor Status Register.
[3]WIM: Window Invalid Mask.
[4]TBF: Trap Base Register.
[5]Y: Extended data register for some arithmetic operations.

6. Initializes hardware and virtual timers.
7. Initializes the scheduler.
8. Initializes the communication channels.
9. Booting partitions are set in NORMAL state and non-booting ones are set in HALT state.
10. Finally, the `setup` function calls the scheduler and becomes into the idle task.

④ Partition code is executed according to the configured plan.

⑤ A system partition can load from PROM or other source (serial line, etc.) the image of other partitions.

⑥ The new ready partition is activated via a `XM_reset_partition()` service.

This page is intentionally left blank.

# Chapter 8

# Configuration

This section describes how XtratuM is configured. There are two levels of configuration. A first level which affects the source code 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`)

The first step in the XtratuM configuration is to configure the source code. This task is done using the same tool than the one used in Linux, which are commonly called "make menuconfig".

There are two different blocks that shall be configured: 1) XtratuM source code; and 2) the resident sofware. The configuration menu of each block is presented one after the other when executed the "`$ make menuconfig`" from the root source directory. The selected configuration are stored in the files `core/.config` and `/user/bootloaders/rsw/.config` for XtratuM and the resident software respectively.

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. Only the options that can be selected are presented to the user.

| Parameter | Type | Default value |
|---|---|---|
| **Processor** | | |
| SPARC cpu | choice | [Leon2] [Leon3] [Leon3FT] [Leon4] |
| Board | choice | [TSim] [GR-CPCI-XC4V] [GR-PCI-XC2V] [GR712RC] [CPU-ITAR-FREE] [SpW-RTC] [SIMLEON] [GR-CPCI-XC4VLX200] |
| Multicore support | choice | [None] [AMP support] [SMP support] |
| Number of CPUs supported | int | 4 if (GR_CPCI_XC4VLX200) |
| SPARC memory protection schema | choice | [MPU (Write Protection Registers)] [MMU] |
| Enable watchdog | bool | n |
| Reset watchdog every N MAFs | int | 1 |

*Continues...*

| Parameter | Type | Default value |
|---|---|---|
| Watchdog timeout | int | 5000000 |
| EDAC support | bool | y |
| Enable cache | bool | y |
| Enable cache snoop | bool | n |
| Enable instruction burst fetch | bool | n |
| Flush cache after context switch | bool | n |
| Select L2 cache write policy | choice | [Copy-back] [Write-through] |
| Enable Power-Down | bool | y |
| **Physical memory layout** | | |
| XM load address | hex | |
| XM virtual address | hex | |
| Enable experimental features | bool | n |
| Enable assertions | bool | y |
| Debug and profiling support | bool | y |
| Dump CPU state when a trap is raised | bool | y |
| Max. identifier length (B) | int | 16 |
| Max. number of custom files | int | 3 |
| **Hypervisor** | | |
| Enable voluntary preemption support | bool | n |
| Kernel stack size (KB) | int | 8 |
| Number of virtual CPUs | int | 2 |
| Number of IPVIs | int | 4 |
| Enable external synchronisation | bool | n |
| Enable kernel audit events | bool | n |
| Compress kernel image | bool | n |
| Jump to user function on cold reset system | bool | n |
| **MMU** | | |
| **Drivers** | | |
| Reserve UART1 | bool | n |
| Reserve UART2 | bool | n |
| Enable early output | bool | n |
| CPU frequency (MHz) | int | |
| Early UART baudrate | int | |
| Select early UART port | choice | [UART1] [UART2] |
| Enable UART flow control | bool | n |
| DSU samples UART port | bool | n |
| GR712 Watchdog Driver | bool | n |
| Enable MIL-STD-1553 as MAF Synchronization | bool | n |
| Remote Terminal Address | int | 20 |
| Descriptor Memory Address | hex | 0x40700000 |
| Enable odd parity | bool | y |
| Select clock source | choice | [Internal clock (24Mhz)] [External clock 12Mhz] [External clock 16Mhz] [External clock 20Mhz] [External clock 24Mhz] |
| Enable LICE support | bool | n |
| Trace internal XM events on LICE | bool | n |
| Trace scheduling XM events on LICE | bool | n |

*Continues...*

| Parameter | Type | Default value |
|---|---|---|
| LICE memory address | hex | 0x2C009000 |
| **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 |
| Maximum number of communication ports | int | 256 |

**Sparc cpu:** Processor model.

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

**Multicore support:** Enables the support for multicore either AMP or SMP.

**Number of CPUs supported:** Configures the maximum number of CPUs supported by the system.

**SPARC memory protection schema:** Select the processor mechanism that will be used to implement memory protection. If MMU is available then it is the best choice. With MPU, only write protection can be enforced.

**Enable watchdog:** When available, this option enables the use of the board's watchdog.

**Reset watchdog every N MAFs:** If the watchdog is enabled, it is reseted every N MAFs.

**Watchdog timeout:** Value used to arm the watchdog.

**Enable EDAC support:** When available, enables the EDAC support for GR712RC.

**Enable cache:** If selected, the processor cache is enabled. All partitions will be executed with cache enabled unless explicitly disabled on each partition through the XM_CF file.

If this option is not selected, the cache memory is disabled. Neither XtratuM nor the partitions will be able to use the cache.

**Enable cache snoop:** Enables cache snooping.

**Enable instruction burst fetch:** Enables the cache instruction burst fetch.

Note that this feature in the LEON processor, this feature enables/disables the cpu stream mode, having a deep impact on the global system performance.

**Flush cache after context switch:** Forces a cache flush after a partition context switch.

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

**XtratuM virtual address:** Virtual address where XtratuM shall be located in the virtual memory map.

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

**Enable assertions:** XtratuM is compiled with robutness assertions enabled.

**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.

**Dump CPU state when a trap is raised:** XtratuM outputs in console the state of the system when a new trap is raised.

**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.

**Enable voluntary preemption support:** The kernel allows to be preempted in a very controlled way.

**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 external synchronisation:** Enables the support of an external timing source to synchronise the execution of the scheduling plans.

**Compress kernel image:** Enables the compression of the kernel XEF file using the LZSS algorithm.

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

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

**Size of the XM's internal buffer to store trace logs:** Maximum number of trace logs stores before discarding.

**Size of the XM's internal buffer to store hm logs:** Maximum number of hm logs stores before discarding.

**Enable UART driver:** Enables XtratuM UART device support.

**Reserve UART1:** Enables the use of UART1 by part of XtratuM.

**Reserve UART2:** Enables the use of UART1 by part of XtratuM.

**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.

**CPU frequency (MHz):** CPU frequency used by early output.

Depending on the device used as early ouput, the CPU frequency could be required. However, at this early stage, neither AMBA Pnp nor XM_CF has been parsed.

**Early UART baudrate:** If early output is enabled and a UART port has been selected as early output device, then this baudrate is the one used to configure the UART.

**Select early UART port:** If early output is enabled this option enables the user to select either UART1 or UART2 as the early default output.

**Enable UART flow control:** Enables UART flow data control.

**DSU samples UART port:** If the UART port used to print console messages is also used by the DSU (Debugging Support Unit), then this option shall be set.

If the DSU is used, then the control bits of the UART does not change. In this case, bytes are sent with a timeout.

**Enable memory block driver:** Enables the use of memory areas as a kind of RAM storage area.

**Enable MIL-STD-1553:** Enables the MIL-STD-1553 driver.

**Select Clock source:** If MIL-STD-1553 driver is enabled, this option allows to configure the clock source of the device.

The default option is the internal timer at 24 MHz. External clock sources are also available at 12, 16, 20 and 24Mhz. MIL-1553 uses the hardware interrupt 14.

**Remote terminal address:** RT address for the core.

**Descriptor memory address:** Each 1553B RT has a reserved location in memory for storing information on how to process various subaddresses and mode codes. The memory space is referred to as the Descriptor Table. This descriptor memory address contains the address that points to the top of this reserved memory space. The memory size should be at least 128KB.

**Enable odd parity:** Enables the odd parity in MIL-STD-1553 transmissions. Otherwise, the even parity is enabled.

**Enable LICE support:** Enables the use of LICE tracing device.

**Trace internal XM events on LICE:** Some internal events (traps arrival, ...) are traced through the LICE.

**Trace scheduling XM events on LICE:** Some XtratuM scheduling events (context switch start, context switch end, ...) are traced through the LICE.

**LICE memory address:** Memory address used by the LICE device.

**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.

**Enable UART support:** If enabled, XtratuM will use the UART to output console messages; otherwise the UART can be used by a partition.

**Enable XM/partition status accounting:** Enable this option to colled statistical information of XtratuM itself and partitions.

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

## 8.2 Resident software source code configuration (menuconfig)

The resident software (RSW) configuration parameters are hard-coded in the source code in order to generate a self-contained stand alone executable code.

After the configuration of the XtratuM source code, the "`$ make menuconfig`" shows the RWS configuration menu. The selected configuration is stored in the file `user/bootloaders/rsw/.config`.

The following parameters can be configured:

| Parameter | Type | Default value |
|---|---|---|
| **RSW memory layout** | | |
| Container physical location address | hex | 0x4000 |
| Read-only section addresses | hex | 0x40200000 |
| Read/write section addresses | hex | 0x40090000 |
| CPU frequency (KHz) | int | 50000 |
| Enable UART support | choice | [UART1] [UART2] |
| Enable UART flow control | bool | n |
| UART baud rate | int | 115200 |
| Stack size (KB) | int | 8 |
| Stand-alone version | bool | no |
| Load container at a fixed address | bool | y |
| Update partition entry point with XEF | bool | n |

**Stack size (KB):** Size of the stack used by the RSW.

**Read-only section addresses:** RSW memory layout. Read-only area. The resident software will be compiled to use this area.

**Read/write section addresses:** RSW memory layout. Read/write area used by the resident software.

**CPU frequency (KHz):** The **processor frequency** is passed to XtratuM via the `XM_CF` file, but in the case of the RSW it has to be specified in the source code, since it has no run-time configuration. The processor frequency is used to configure the UART clock.

**Enable UART support:** Select the serial line to write messages to.

**UART baud rate:** The baud rate of the UART. Note that the baud rate used by XtratuM is configured in the `XM_CF` file, and not in the source code configuration.

**Stand-alone version:** If not set, then the resident software shall be linked jointly with the container. That is, the final resident software image shall contain, as data, the container.

If set, then the container is not linked with the resident software. The address where the container will be copied in RAM is specified by the next option:

**Container physical location address:** Address of the container. In case of the stand-alone version.

### 8.2.1 Memory requirements

The memory footprint of XtratuM is independent of the workload (number of partitions, channels, etc.) The memory needed depends only on actual workload defined in the `XM_CF` file. The size of the compiled configuration provides an accurate estimation of the memory what will use XtratuM to run it. Note that it is not the size of the file, but the memory required by all the sections (including those not allocatable ones: `.bss`).

The resident software can be executed in ROM or RAM memory (if the ROM technology allows to run eXecute-in-Place XiP code). The resident software has no initialised data segment, only the `.text` and `.rodata` segments are required (the `.got` and `.eh_frame` segments are not used). The memory footprint of the RSW depends on whether the debug messages are enabled or not; and it can be obtained from the final resident software code (the file created with the `build_rsw` helper utility) using the `readelf` utility.

The next example shows where the size of the RSW code is printed (column labeled **`MemSiz`**):

```
# sparc-linux-readelf -l resident_sw
Elf file type is EXEC (Executable file)
Entry point 0x4020102c
There are 5 program headers, starting at offset 52

Program Headers:
  Type          Offset    VirtAddr   PhysAddr  FileSiz MemSiz  Flg  Align
  LOAD          0x0000d4 0x00004000 0x00004000 0x196f0 0x196f0 RW 0x1 // 00 segment
  LOAD          0x0197c4 0x40090000 0x0001d6f0 0x00048 0x00048 RW 0x4 // 01 segment
  LOAD          0x019808 0x40090048 0x40090048 0x00000 0x02000 RW 0x8 // 02 segment
  LOAD          0x019810 0x40200000 0x40200000 0x01f44 0x01f44 R E 0x8 // 03 segment
  GNU_STACK     0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE 0x4 // 04 segment

 Section to Segment mapping:
  Segment Sections...
   00     .container
   01     .got .eh_frame
   02     .bss
   03     .text .rodata
```

```
04
```

Listing 8.1: Resident software memory footprint example.

The next table summarises the size of the resident software for different configurations:

| Board type | Debug messages | size |
|---|---|---|
| LEON3 | disabled | 0x01f44 |
| LEON3 | enabled | 0x01ff4 |
| LEON2 | disabled | 0x01f44 |
| LEON2 | enabled | 0x01ff4 |

## 8.3 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 attributed, and how many times an element can appear) the reader is referred to the XML schema definition in the Appendix A.

### 8.3.1 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])"
Examples of valid times:

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

**Size:** Pattern: "[0-9]+(.[0-9]+)?([MK]?B)"
Examples of valid sizes:

```
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.
```

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.
```

**Boolean:** Valid values are: "yes", "true", "no", "false".

**Hexadecimal:** Pattern: "0x[0-9a-fA-F]+"
Examples of valid numbers:

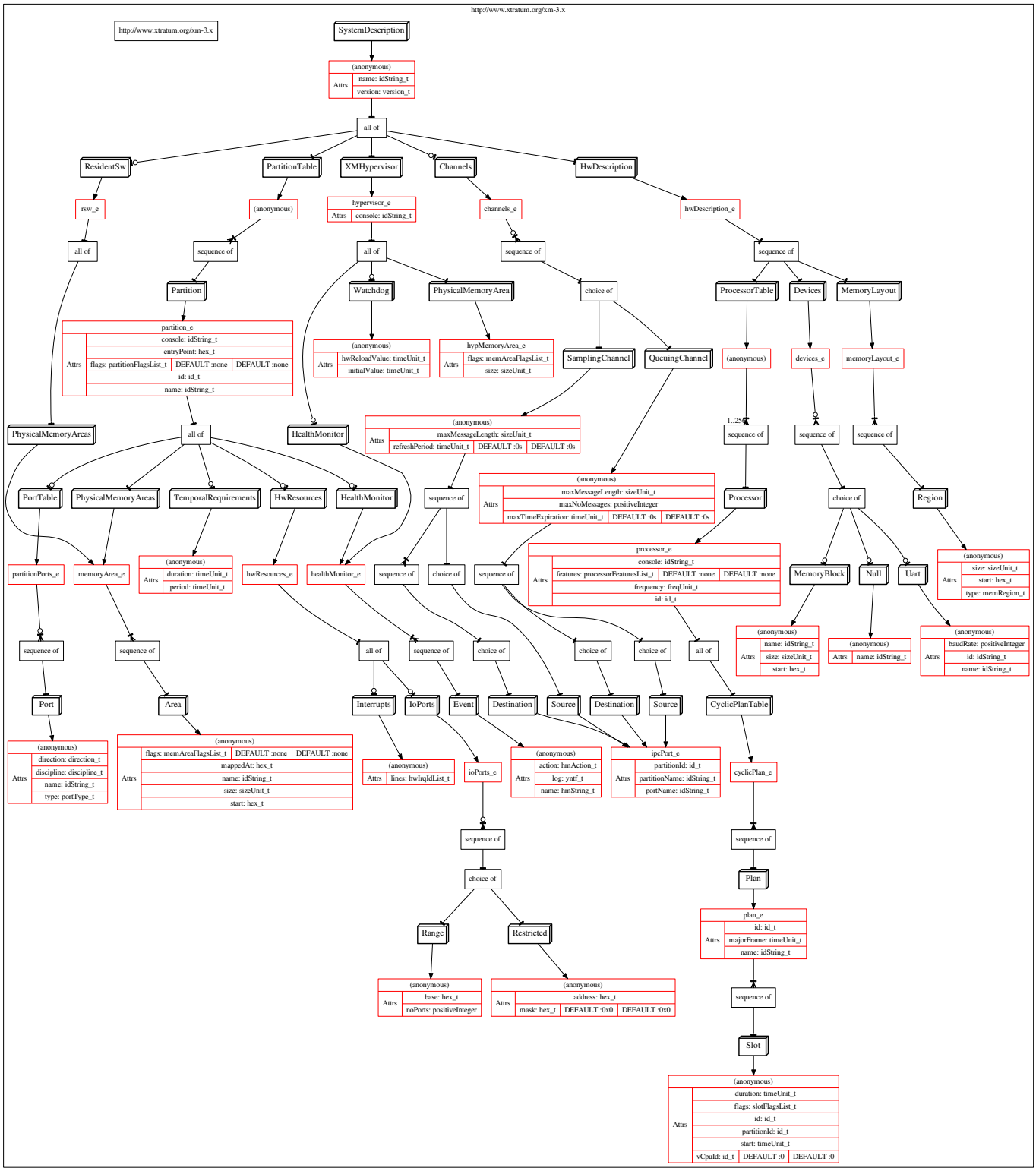



Figure 8.1: XML Schema diagram.

```
0xFFFFFFFF, 0x0, 0xF1, 0x80
```

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**.

### 8.3.2 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.

Figure 8.2 on page 106 is a compact graphical representation of the nested structure a sample XM_CF configuration file (the listing A.2 is the actual xml file for this representation). Solid-lined boxes represent elements. Dotted boxes contain attributes. The nested boxes represent the hierarchy of 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-2.3".

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 which for providing information to XtratuM about the resident software.

**/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** Contain the configuration of physical and virtual resources.

### 8.3.3 The /SystemDescription/XMHypervisor element

There is one optional attribute **./@console**. The values of this attribute shall be the name of a device defined in the **/SystemDescription/HwDescription/Devices** section.

Mandatory elements:

**./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:

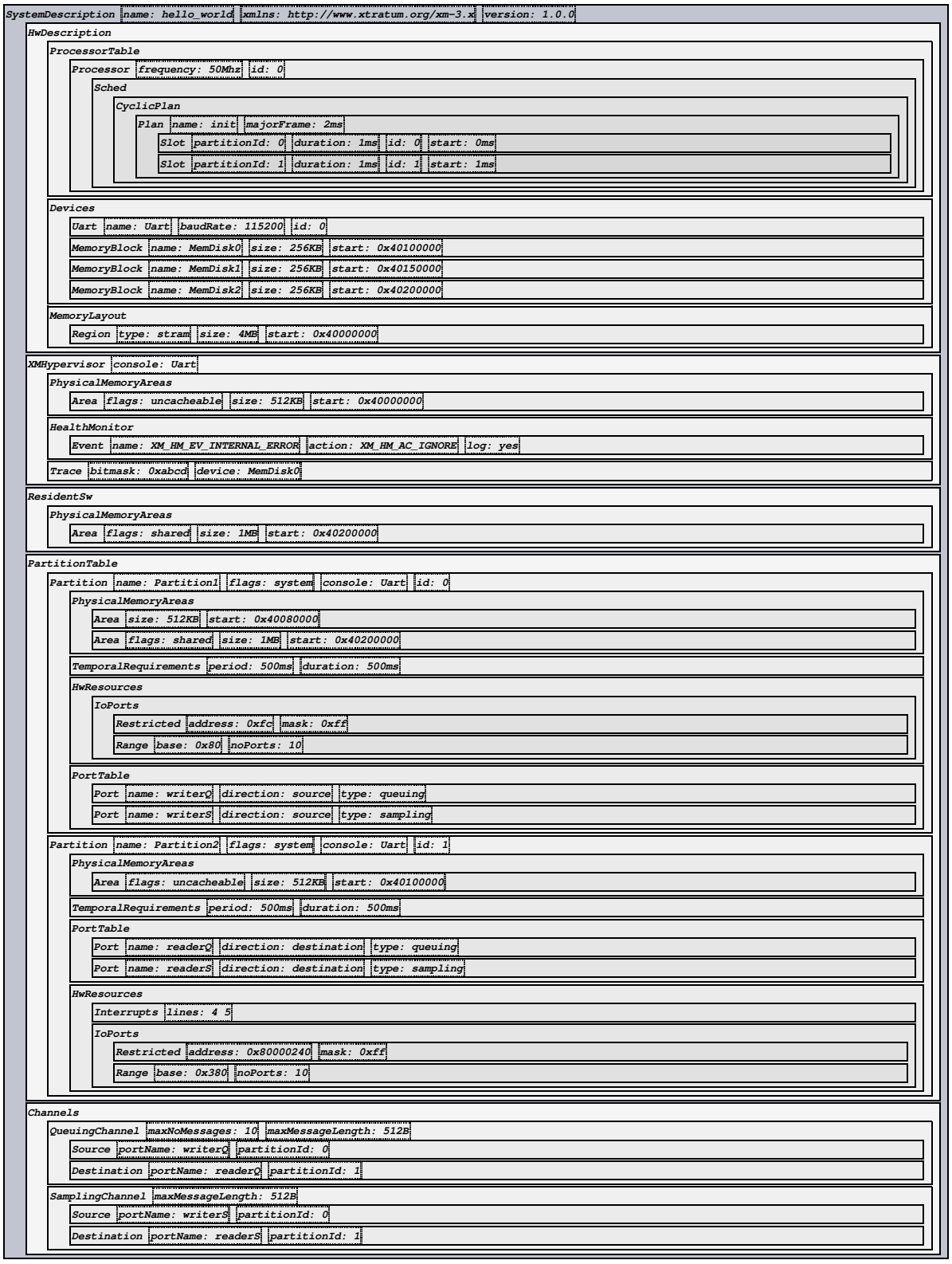



Figure 8.2: Graphical view of an example XM_CF configuration file (see the XML file in section A.2).

**./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_UNRECOVERABLE"/>
      <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"/>
      <xs:enumeration value="XM_HM_EV_EXTSYNC_ERROR"/>
#ifdef CONFIG_SPARCv8
      <xs:enumeration value="XM_HM_EV_SPARCV8_WRITE_ERROR"/>
      <xs:enumeration value="XM_HM_EV_SPARCV8_INSTR_ACCESS_MMU_MISS"/>
      <xs:enumeration value="XM_HM_EV_SPARCV8_INSTR_ACCESS_ERROR"/>
      <xs:enumeration value="XM_HM_EV_SPARCV8_UNIMPLEMENTED_FLUSH"/>
      <xs:enumeration value="XM_HM_EV_SPARCV8_WATCHPOINT_DETECTED"/>
      <xs:enumeration value="XM_HM_EV_SPARCV8_DATA_ACCESS_ERROR"/>
      <xs:enumeration value="XM_HM_EV_SPARCV8_DATA_ACCESS_MMU_MISS"/>
      <xs:enumeration value="XM_HM_EV_SPARCV8_INSTR_ACCESS_EXCEPTION"/>
      <xs:enumeration value="XM_HM_EV_SPARCV8_ILLEGAL_INSTR"/>
      <xs:enumeration value="XM_HM_EV_SPARCV8_PRIVILEGED_INSTR"/>
      <xs:enumeration value="XM_HM_EV_SPARCV8_FP_DISABLED"/>
      <xs:enumeration value="XM_HM_EV_SPARCV8_CP_DISABLED"/>
      <xs:enumeration value="XM_HM_EV_SPARCV8_REGISTER_HARDWARE_ERROR"/>
      <xs:enumeration value="XM_HM_EV_SPARCV8_MEM_ADDR_NOT_ALIGNED"/>
      <xs:enumeration value="XM_HM_EV_SPARCV8_FP_EXCEPTION"/>
      <xs:enumeration value="XM_HM_EV_SPARCV8_CP_EXCEPTION"/>
      <xs:enumeration value="XM_HM_EV_SPARCV8_DATA_ACCESS_EXCEPTION"/>
      <xs:enumeration value="XM_HM_EV_SPARCV8_TAG_OVERFLOW"/>
      <xs:enumeration value="XM_HM_EV_SPARCV8_DIVIDE_EXCEPTION"/>
      <xs:enumeration value="XM_HM_EV_SPARCV8_UNCORRECTABLE_EDAC_ERROR"/>
#endif
      <xs:enumeration value="XM_HM_EV_APP_DEADLINE_MISSED"/>
      <xs:enumeration value="XM_HM_EV_APP_APPLICATION_ERROR"/>
      <xs:enumeration value="XM_HM_EV_APP_NUMERIC_ERROR"/>
      <xs:enumeration value="XM_HM_EV_APP_ILLEGAL_REQUEST"/>
      <xs:enumeration value="XM_HM_EV_APP_STACK_OVERFLOW"/>
      <xs:enumeration value="XM_HM_EV_APP_MEMORY_VIOLATION"/>
      <xs:enumeration value="XM_HM_EV_APP_HARDWARE_FAULT"/>
      <xs:enumeration value="XM_HM_EV_APP_POWER_FAIL"/>
    </xs:restriction>
  </xs:simpleType>
```

Listing 8.2: 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 8.3: user/tools/xmcparser/xmc.xsd.in

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

### 8.3.4 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), `./@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 bug[1]. 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.

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`, `./@majorFrame`, `./@extSync` and `./@interval`; and contains a sequence of `./Slot` elements.

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

`./Slot/@id` Slot Id's shall meet the id's rules defined in section 2.4. This value can be retrieved by the partition at run time, see section 5.7.1.

`./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.

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.

`./HwDescription/Devices` The devices element contains the sequence the 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:

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

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

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

[1] Some 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.

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

### 8.3.5 The /SystemDescription/ResidentSw element

The element **./PhysicalMemoryAreas** is used to declare the memory areas where the resident software will by 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 overlaps errors.

### 8.3.6 The /SystemDescription/PartitionTable/Partition element

Attribute description:

**./@id** Required. See the 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. By default not set.
- **boot** If true, 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. By default not set.

**./@entryPoint** Entry point of the partition. XtratuM starts the execution of the partition at this address.

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.
- **./Trace** Configuration of the trace facility of the partition. Same attributes than that of the /SystemDescription/XMHypervisor/Trace 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. |
| rom | Not applicable in SPARC v8 boards. Only used in ia32 systems. |

#### Configuration of I/O ports

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).

`./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.
  Those bits not allocated to this partition (i.e. the bit not set in the bitmask) can be allocated to other partitions.

#### Configuration of interrupts

The element `./Partition/HwResources/Interrupts` has the attribute `./@lines` which is a list of the interrupt number (in the range 0 to 16) allocated to the partition.

### 8.3.7 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:

- `./@maxMessageLength` 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:

- `./@maxMessageLength` 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. Thefore, 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.

This page is intentionally left blank.

# Chapter 9

# Internal instrumentation

## 9.1 Hardware Instrumentation

### 9.1.1 Interface to the LICE analiser

If selected in the menuconfig, XtratuM traces in the LICE device some relevant events. Those traced events are grouped in two categories:

**Internal operation:** If the "Trace internal XM events on LICE" option is selected in the menuconfig, then the following events are recorded:

`LICE_CODE_START_XM`: Marks the start of the XtratuM initialisation.

`LICE_CODE_BEGIN_TRAP`: Marks the start of:

- Hardware interrupts.
- Normal hypercalls.
- Processor exceptions.

`LICE_CODE_END_TRAP`: Marks the end of a `LICE_CODE_BEGIN_TRAP` event.

`LICE_CODE_BEGIN_CS`: Marks the start of a context switch.

`LICE_CODE_END_CS`: Marks the end of a context switch.

`LICE_CODE_BEGIN_FTRAP`: Marks the start of fast trap:

- Clock hardware interrupt.
- Fast hypercalls.

`LICE_CODE_END_FTRAP`: Marks the end of `LICE_CODE_BEGIN_FTRAP` event.

`LICE_CODE_HYPERCALL_NR`: The number of the hypercall is recorded as LICE data. This event is recorded after the `LICE_CODE_END_FTRAP` event.

`LICE_CODE_ASM_HYPERCALL_NR`: Marks the start of an assembly hypercall. The number of the hypercall is recorded as LICE data.

`LICE_CODE_END_ASM_HYPERCALL`: Marks the end of an assembly hypercall.

**Scheduling events:** If the "Trace scheduling events on LICE" option is selected in the menuconfig, then the following events are recorded:

`LICE_CODE_START_PARTITION`: Marks the start of execution of the next partition. The data recorded with this is event is:

**bits 31:16** Id of the new partition.

**bits 15:0** Slot id number.

This event is emitted right after the END_PARTITION one. And the

LICE_CODE_END_PARTITION: Marks the end of execution of the current partition. The data recorded with this is event is:

**bits 31:16** Id of the ending partition.

**bits 15:0** Slot id number.

The first 24 codes (I/O addresses) of the LICE are reserved by XtratuM if configured to use the LICE. The rest of the LICE codes can be used (allocated to) any partition. The partitions may access to the LICE I/O ports using the standard XM_sparcv8_outport() hypercall.

### 9.1.2 Partition instrumentation

Those LICE ports not used by XtratuM can be used allocated by the partition (see the textttcore/include/sparcv8/ginfo.h file for a list of the I/O ports used by XtratuM). The ports to be used by the partition shall be allocated to it in the XM_CF file and use the XM_sparcv8_outport() hypercall to write on them.

## 9.2 Software Instrumentation

When the "Enable kernel audit events" option is enabled in XtratuM source code configuration (menuconfig), XtratuM logs the following events as traces:

XM_AUDIT_START_WATCHDOG : Marks the start of the programming of the watchdog.

XM_AUDIT_END_WATCHDOG : Marks the end of the programming of the watchdog.

XM_AUDIT_BEGIN_PARTITION : Marks the beginning of the execution of the next partition. This event is generated right after the event indicating the end of the previous partition (or the idle thread) execution.

XM_AUDIT_END_PARTITION : Marks the end of the execution of the current partition.

XM_AUDIT_BEGIN_CS : Marks the beginning of a context switch.

XM_AUDIT_END_CS : Marks the end of a context switch.

XM_AUDIT_BEGIN_IDLE : Marks the beginning of the execution of the idle thread. This event is generated right after the event indicating the end of the previous partition execution.

XM_AUDIT_END_IDLE : Marks the end of the idle thread execution.

These traces can later be read by a system partition through the XM_trace_read() hypercall. The partition identifier related to the event can be found in the first position of the payload member of each retrieved xmTraceEvent_t struct. Likewise, the event identifier is located in the second position of the payload member.

# Chapter 10

# 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.

**`xmpack`:** Creates the container file.

**`rswbuild`:** Creates a bootable file image.

**`ampbuilder.py`:** Creates a bootable file image from several XtratuM instances.

## 10.1 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:

- 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.

### 10.1.1 xmcparser

Compiles XtratuM XML configuration files

#### SYNOPSIS

**xmcparser** [-c] [-e] [-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 folowing 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 dafault XML schema used to validate the XML configuration file.

**-o file**

Place output in file.

**-e**

Prints the size required in the target for the configuration.

**-s xsd_file**

Use the XML schema xsd_file rather than the dafault XtratuM schema.

**-c**

Stop after the stage of ”C” generation; do not compile. The output is in the form of a ”C” file.

## 10.2 ELF to XEF (xmeformat)

### 10.2.1 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.

**-o file**

Places output in file file.

**-c**

The XEF segments are compressed using the LSZZ algorithm.

**-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**

Shows the contents of the XEF file.

**-h**

Print the content of the header.

**-s**

Lists the segments and its attributes.

**-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
```

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
```

Build the hypervisor XEF file:

```
$ xmeformat build -o xm_core.xef -c core/xm_core
```

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
XEF header:
  signature: 0x24584546
  version: 1.0.0
  flags: XEF_DIGEST XEF_COMPRESSED XEF_CONTENT_HYPERVISOR
  digest: 6698cfcf9311325e46e79ed50dfc9683
  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
  customFile table offset: 104
  no. customFiles: 1
  image offset: 112
  image length: 68520
  XM image's header: 0x40010b78
  compressed image length: 32928 (48.06%)
```

## 10.3 Container builder (xmpack)

### 10.3.1 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

**xmpack list** -c container

#### DESCRIPTION

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.

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).
- 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).

**-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.

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.

**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**

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
```

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>
```

## 10.4 Bootable image creator (`rswbuild`)

### 10.4.1 rswbuild

Create a bootable image

**SYNOPSIS**

**rswbuild** contailer bootable

#### 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

```
rswbuild container resident_sw
```

## 10.5 AMP Bootable image creator (ampbuilder.py)

### 10.5.1 ampbuilder

Create an AMP bootable image

#### SYNOPSIS

**ampbuilder.py** loader rsw0 rsw1

#### DESCRIPTION

ampbuilder is a python script that creates a bootable file by combining several resident sofwares with a bootloader. When the resulting file is loaded into the board, the bootloader sets up the processors, allocating each one to a XtratuM instance. The resulting file is called image.

#### USAGE EXAMPLES

```
ampbuilder ../../tools/ampbuilder/boot.o resident_sw0 resident_sw1
```

This page is intentionally left blank.

# Chapter 11

# Security issues

This chapter introduces several security issues related with XtratuM which should be taken into account by partition developers.

## 11.1 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.

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.

## 11.2 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:

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.

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.

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 lest some idle time before the end of a slot and the beginning of the next one. The Xoncrete scheduling tool is able to implement that solution automatically. Figure 11.1 shows two scenarios: scenario 1 where the integrator has not left spare time between one partition slot and the next one, enabling the partition in light grey overrunning 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 end occurs, so the next partition will start always with no delay.
3. At hypervisor level:
   (a) Change the design of XtratuM to convert 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.

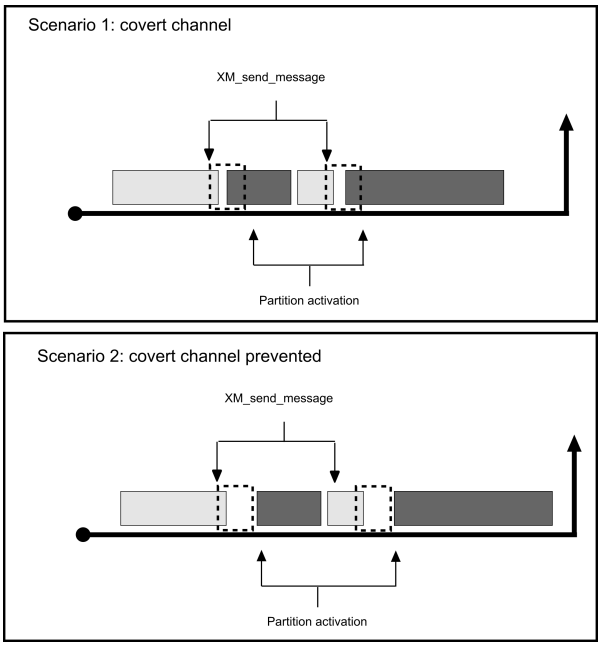



Figure 11.1: Covert channel caused by an incorrect scheduling plan and a solution.

# Chapter 12

# Known limitations

This is the list of currently known limitations:

- In SMP systems, queuing/sampling ports do not update the interrupt bitmap.
- `XM_get_partition_status()` does not update the `noVIrqs` field of `xmPartitionStatus_t`.
- For time values of margins between slots in the scheduling plan that are inferior to the context switch time, it may happen that the timer signaling the beginning of the next slot expires exactly while being armed. In this case the idle thread is already selected by the scheduler, that will complete the context switch and reschedule the system as soon as the context switch finishes. This will add an overhead to the context switch time that must be taken into account by the system integrator.

This page is intentionally left blank.

# Appendix A

# XML Schema Definition

## A.1 XML Schema file

basicstyle

```
<?xml version="1.0"?>
<xs:schema targetNamespace="http://www.xtratum.org/xm-3.x"
          xmlns:xs="http://www.w3.org/2001/XMLSchema"
          xmlns="http://www.xtratum.org/xm-3.x"
          elementFormDefault="qualified"
          attributeFormDefault="unqualified">
<!--
     Changelog:
     - [02/09/2015:SPR-280715-01]: entryPoint added to partition_e definition
     - [18/11/2015:SPR-221015-01] edacCounter added to xmSystemStatus_t
     - [20/11/2015:SPR-110915-01] ipvi stuff depends on IPVI_SUPPORT
     - [03/02/2016:SPR-111215-03] XM_HM_EV_PARTITION_UNRECOVERABLE missed.
     - [30/03/2016:CP-250216-01] ngel Esquinas (aesquinas@fentiss.com)
       added "Watchdog" as a child node of "Hypervisor". The node define
       the attributes "initialValue" and "hwReloadValue".
-->
  <!-- Basic types definition -->
  <xs:simpleType name="id_t">
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="0"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="idString_t">
    <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="hwIrqId_t">
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="0"/>
      <xs:maxExclusive value="
                            16
                            "/>
    </xs:restriction>
```

```
35    </xs:simpleType>
36    <xs:simpleType name="hwIrqIdList_t">
37      <xs:list itemType="hwIrqId_t"/>
38    </xs:simpleType>
39    <xs:simpleType name="idList_t">
40      <xs:list itemType="id_t"/>
41    </xs:simpleType>
42    <xs:simpleType name="hex_t">
43      <xs:restriction base="xs:string">
44        <xs:pattern value="0x[0-9a-fA-F]+"/>
45      </xs:restriction>
46    </xs:simpleType>
47    <xs:simpleType name="version_t">
48      <xs:restriction base="xs:string">
49        <xs:pattern value="[0-9]+.[0-9]+.[0-9]+"/>
50      </xs:restriction>
51    </xs:simpleType>
52    <xs:simpleType name="freqUnit_t">
53      <xs:restriction base="xs:string">
54        <xs:pattern value="[0-9]+(.[0-9]+)?([MK][Hh]z)"/>
55      </xs:restriction>
56    </xs:simpleType>
57    <xs:simpleType name="processorFeatures_t">
58      <xs:restriction base="xs:string">
59        <xs:enumeration value="XM_CPU_LEON2_WA1"/>
60        <xs:enumeration value="none"/>
61      </xs:restriction>
62    </xs:simpleType>
63    <xs:simpleType name="discipline_t">
64      <xs:restriction base="xs:string">
65        <xs:enumeration value="FIFO"/>
66        <xs:enumeration value="PRIORITY"/>
67      </xs:restriction>
68    </xs:simpleType>
69    <xs:simpleType name="processorFeaturesList_t">
70      <xs:list itemType="processorFeatures_t"/>
71    </xs:simpleType>
72    <xs:simpleType name="partitionFlags_t">
73      <xs:restriction base="xs:string">
74        <xs:enumeration value="system"/>
75        <xs:enumeration value="fp"/>
76        <xs:enumeration value="boot"/>
77        <xs:enumeration value="icache_disabled"/>
78        <xs:enumeration value="dcache_disabled"/>
79        <xs:enumeration value="none"/>
80      </xs:restriction>
81    </xs:simpleType>
82    <xs:simpleType name="partitionFlagsList_t">
83      <xs:list itemType="partitionFlags_t"/>
84    </xs:simpleType>
85    <xs:simpleType name="sizeUnit_t">
86      <xs:restriction base="xs:string">
87        <xs:pattern value="[0-9]+(.[0-9]+)?([MK]?B)"/>
88      </xs:restriction>
```

```
 89    </xs:simpleType>
 90    <xs:simpleType name="timeUnit_t">
 91      <xs:restriction base="xs:string">
 92        <xs:pattern value="[0-9]+(.[0-9]+)?([mu]?[sS])"/>
 93      </xs:restriction>
 94    </xs:simpleType>
 95
 96    <xs:simpleType name="hmString_t">
 97      <xs:restriction base="xs:string">
 98        <xs:enumeration value="XM_HM_EV_INTERNAL_ERROR"/>
 99        <xs:enumeration value="XM_HM_EV_UNEXPECTED_TRAP"/>
100        <xs:enumeration value="XM_HM_EV_PARTITION_ERROR"/>
101        <xs:enumeration value="XM_HM_EV_PARTITION_UNRECOVERABLE"/>
102        <xs:enumeration value="XM_HM_EV_PARTITION_INTEGRITY"/>
103        <xs:enumeration value="XM_HM_EV_MEM_PROTECTION"/>
104        <xs:enumeration value="XM_HM_EV_OVERRUN"/>
105        <xs:enumeration value="XM_HM_EV_SCHED_ERROR"/>
106        <xs:enumeration value="XM_HM_EV_WATCHDOG_TIMER"/>
107        <xs:enumeration value="XM_HM_EV_INCOMPATIBLE_INTERFACE"/>
108        <xs:enumeration value="XM_HM_EV_EXTSYNC_ERROR"/>
109        <xs:enumeration value="XM_HM_EV_SPARCV8_WRITE_ERROR"/>
110        <xs:enumeration value="XM_HM_EV_SPARCV8_INSTR_ACCESS_MMU_MISS"/>
111        <xs:enumeration value="XM_HM_EV_SPARCV8_INSTR_ACCESS_ERROR"/>
112        <xs:enumeration value="XM_HM_EV_SPARCV8_UNIMPLEMENTED_FLUSH"/>
113        <xs:enumeration value="XM_HM_EV_SPARCV8_WATCHPOINT_DETECTED"/>
114        <xs:enumeration value="XM_HM_EV_SPARCV8_DATA_ACCESS_ERROR"/>
115        <xs:enumeration value="XM_HM_EV_SPARCV8_DATA_ACCESS_MMU_MISS"/>
116        <xs:enumeration value="XM_HM_EV_SPARCV8_INSTR_ACCESS_EXCEPTION"/>
117        <xs:enumeration value="XM_HM_EV_SPARCV8_ILLEGAL_INSTR"/>
118        <xs:enumeration value="XM_HM_EV_SPARCV8_PRIVILEGED_INSTR"/>
119        <xs:enumeration value="XM_HM_EV_SPARCV8_FP_DISABLED"/>
120        <xs:enumeration value="XM_HM_EV_SPARCV8_CP_DISABLED"/>
121        <xs:enumeration value="XM_HM_EV_SPARCV8_REGISTER_HARDWARE_ERROR"/>
122        <xs:enumeration value="XM_HM_EV_SPARCV8_MEM_ADDR_NOT_ALIGNED"/>
123        <xs:enumeration value="XM_HM_EV_SPARCV8_FP_EXCEPTION"/>
124        <xs:enumeration value="XM_HM_EV_SPARCV8_CP_EXCEPTION"/>
125        <xs:enumeration value="XM_HM_EV_SPARCV8_DATA_ACCESS_EXCEPTION"/>
126        <xs:enumeration value="XM_HM_EV_SPARCV8_TAG_OVERFLOW"/>
127        <xs:enumeration value="XM_HM_EV_SPARCV8_DIVIDE_EXCEPTION"/>
128        <xs:enumeration value="XM_HM_EV_SPARCV8_UNCORRECTABLE_EDAC_ERROR"/>
129        <xs:enumeration value="XM_HM_EV_APP_DEADLINE_MISSED"/>
130        <xs:enumeration value="XM_HM_EV_APP_APPLICATION_ERROR"/>
131        <xs:enumeration value="XM_HM_EV_APP_NUMERIC_ERROR"/>
132        <xs:enumeration value="XM_HM_EV_APP_ILLEGAL_REQUEST"/>
133        <xs:enumeration value="XM_HM_EV_APP_STACK_OVERFLOW"/>
134        <xs:enumeration value="XM_HM_EV_APP_MEMORY_VIOLATION"/>
135        <xs:enumeration value="XM_HM_EV_APP_HARDWARE_FAULT"/>
136        <xs:enumeration value="XM_HM_EV_APP_POWER_FAIL"/>
137      </xs:restriction>
138    </xs:simpleType>
139
140
141    <xs:simpleType name="hmAction_t">
142      <xs:restriction base="xs:string">
```

```
143       <xs:enumeration value="XM_HM_AC_IGNORE"/>
144       <xs:enumeration value="XM_HM_AC_SHUTDOWN"/>
145       <xs:enumeration value="XM_HM_AC_PARTITION_COLD_RESET"/>
146       <xs:enumeration value="XM_HM_AC_PARTITION_WARM_RESET"/>
147       <xs:enumeration value="XM_HM_AC_HYPERVISOR_COLD_RESET"/>
148       <xs:enumeration value="XM_HM_AC_HYPERVISOR_WARM_RESET"/>
149       <xs:enumeration value="XM_HM_AC_SUSPEND"/>
150       <xs:enumeration value="XM_HM_AC_HALT"/>
151       <xs:enumeration value="XM_HM_AC_PROPAGATE"/>
152       <xs:enumeration value="XM_HM_AC_SWITCH_TO_MAINTENANCE" />
153     </xs:restriction>
154   </xs:simpleType>
155
156   <xs:simpleType name="memAreaFlags_t">
157     <xs:restriction base="xs:string">
158       <xs:enumeration value="unmapped"/>
159       <xs:enumeration value="read-only"/>
160       <xs:enumeration value="uncacheable"/>
161       <xs:enumeration value="rom"/>
162       <xs:enumeration value="flag0"/>
163       <xs:enumeration value="flag1"/>
164       <xs:enumeration value="flag2"/>
165       <xs:enumeration value="flag3"/>
166       <xs:enumeration value="none"/>
167     </xs:restriction>
168   </xs:simpleType>
169   <xs:simpleType name="memAreaFlagsList_t">
170     <xs:list itemType="memAreaFlags_t"/>
171   </xs:simpleType>
172   <xs:simpleType name="slotFlags_t">
173     <xs:restriction base="xs:string">
174       <xs:enumeration value="periodStart"/>
175     </xs:restriction>
176   </xs:simpleType>
177   <xs:simpleType name="slotFlagsList_t">
178     <xs:list itemType="slotFlags_t"/>
179   </xs:simpleType>
180   <xs:simpleType name="memRegion_t">
181     <xs:restriction base="xs:string">
182       <xs:enumeration value="sdram"/>
183       <xs:enumeration value="stram"/>
184       <xs:enumeration value="rom"/>
185     </xs:restriction>
186   </xs:simpleType>
187   <xs:simpleType name="portType_t">
188     <xs:restriction base="xs:string">
189       <xs:enumeration value="queuing"/>
190       <xs:enumeration value="sampling"/>
191     </xs:restriction>
192   </xs:simpleType>
193   <xs:simpleType name="direction_t">
194     <xs:restriction base="xs:string">
195       <xs:enumeration value="source"/>
196       <xs:enumeration value="destination"/>
```

```
197     </xs:restriction>
198   </xs:simpleType>
199   <xs:simpleType name="yntf_t">
200     <xs:restriction base="xs:string">
201       <xs:enumeration value="yes"/>
202       <xs:enumeration value="no"/>
203       <xs:enumeration value="true"/>
204       <xs:enumeration value="false"/>
205     </xs:restriction>
206   </xs:simpleType>
207   <!-- End Types -->
208   <!-- Elements -->
209   <!-- Hypervisor -->
210   <xs:complexType name="hypervisor_e">
211     <xs:all>
212       <xs:element name="PhysicalMemoryArea" type="hypMemoryArea_e"/>
213       <xs:element name="HealthMonitor" type="healthMonitor_e" minOccurs="0" />
214       <xs:element name="Watchdog" minOccurs="0" maxOccurs="1">
215         <xs:complexType>
216           <xs:attribute name="initialValue" type="timeUnit_t" use="required"/>
217           <xs:attribute name="hwReloadValue" type="timeUnit_t" use="required"/>
218         </xs:complexType>
219       </xs:element>
220     </xs:all>
221     <xs:attribute name="console" type="idString_t" use="optional" />
222   </xs:complexType>
223   <!-- Rsw -->
224   <xs:complexType name="rsw_e">
225     <xs:all>
226       <xs:element name="PhysicalMemoryAreas" type="memoryArea_e"/>
227     </xs:all>
228   </xs:complexType>
229   <!-- Partition -->
230   <xs:complexType name="partition_e">
231     <xs:all>
232       <xs:element name="PhysicalMemoryAreas" type="memoryArea_e"/>
233       <xs:element name="TemporalRequirements" minOccurs="0">
234         <xs:complexType>
235           <xs:attribute name="period" type="timeUnit_t" use="required"/>
236           <xs:attribute name="duration" type="timeUnit_t" use="required"/>
237         </xs:complexType>
238       </xs:element>
239       <xs:element name="HealthMonitor" type="healthMonitor_e" minOccurs="0" />
240       <xs:element name="HwResources" type="hwResources_e" minOccurs="0" />
241       <xs:element name="PortTable" type="partitionPorts_e" minOccurs="0" />
242     </xs:all>
243     <xs:attribute name="id" type="id_t" use="required"/>
244     <xs:attribute name="name" type="idString_t" use="optional" />
245     <xs:attribute name="console" type="idString_t" use="optional" />
246     <xs:attribute name="flags" type="partitionFlagsList_t" use="optional"
          default="none" />
247     <xs:attribute name="entryPoint" type="hex_t" use="required" />
248   </xs:complexType>
249   <!-- Communication Ports -->
```

```
250   <xs:complexType name="partitionPorts_e">
251     <xs:sequence minOccurs="0" maxOccurs="unbounded">
252       <xs:element name="Port">
253         <xs:complexType>
254           <xs:attribute name="name" type="idString_t" use="required"/>
255           <xs:attribute name="direction" type="direction_t" use="required"/>
256           <xs:attribute name="type" type="portType_t" use="required"/>
257           <xs:attribute name="discipline" type="discipline_t" use="optional" />
258         </xs:complexType>
259       </xs:element>
260     </xs:sequence>
261   </xs:complexType>
262   <!-- Channels -->
263   <xs:complexType name="channels_e">
264     <xs:sequence minOccurs="0" maxOccurs="unbounded">
265       <xs:choice>
266         <xs:element name="SamplingChannel">
267           <xs:complexType>
268             <xs:sequence minOccurs="1">
269               <xs:choice>
270                 <xs:element name="Source" type="ipcPort_e" />
271               </xs:choice>
272               <xs:sequence minOccurs="1" maxOccurs="unbounded">
273                 <xs:choice>
274                   <xs:element name="Destination" type="ipcPort_e"/>
275                 </xs:choice>
276               </xs:sequence>
277             </xs:sequence>
278             <xs:attribute name="maxMessageLength" type="sizeUnit_t" use="
                    required"/>
279             <xs:attribute name="refreshPeriod" type="timeUnit_t" use="optional"
                    default="0s"/>
280           </xs:complexType>
281         </xs:element>
282         <xs:element name="QueuingChannel">
283           <xs:complexType>
284            <xs:sequence minOccurs="1">
285               <xs:choice>
286                 <xs:element name="Source" type="ipcPort_e" />
287               </xs:choice>
288               <xs:choice>
289                 <xs:element name="Destination" type="ipcPort_e" />
290               </xs:choice>
291             </xs:sequence>
292             <xs:attribute name="maxMessageLength" type="sizeUnit_t" use="
                    required"/>
293             <xs:attribute name="maxNoMessages" type="xs:positiveInteger" use="
                    required"/>
294              <xs:attribute name="maxTimeExpiration" type="timeUnit_t" use="
                     optional" default="0s"/>
295           </xs:complexType>
296         </xs:element>
297       </xs:choice>
298     </xs:sequence>
```

```
299   </xs:complexType>
300   <!-- Devices -->
301   <xs:complexType name="devices_e">
302     <xs:sequence minOccurs="0" maxOccurs="unbounded">
303       <xs:choice>
304         <xs:element name="MemoryBlock" minOccurs="0">
305           <xs:complexType>
306             <xs:attribute name="name" type="idString_t" use="required"/>
307             <xs:attribute name="start" type="hex_t" use="required"/>
308             <xs:attribute name="size" type="sizeUnit_t" use="required"/>
309           </xs:complexType>
310         </xs:element>
311         <xs:element name="Uart" minOccurs="0">
312           <xs:complexType>
313             <xs:attribute name="name" type="idString_t" use="required"/>
314             <xs:attribute name="id" type="idString_t" use="required"/>
315             <xs:attribute name="baudRate" type="xs:positiveInteger" use="
                    required"/>
316           </xs:complexType>
317         </xs:element>
318         <xs:element name="Null" minOccurs="0">
319           <xs:complexType>
320             <xs:attribute name="name" type="idString_t" use="optional" />
321           </xs:complexType>
322         </xs:element>
323       </xs:choice>
324     </xs:sequence>
325   </xs:complexType>
326   <!-- IPC Port -->
327   <xs:complexType name="ipcPort_e">
328     <xs:attribute name="partitionId" type="id_t" use="required"/>
329     <xs:attribute name="partitionName" type="idString_t" use="optional" />
330     <xs:attribute name="portName" type="idString_t" use="required"/>
331   </xs:complexType>
332   <!-- Hw Description -->
333   <xs:complexType name="hwDescription_e">
334     <xs:sequence>
335       <xs:element name="MemoryLayout" type="memoryLayout_e"/>
336       <xs:element name="ProcessorTable">
337         <xs:complexType>
338           <xs:sequence minOccurs="1" maxOccurs="256">
339             <xs:element name="Processor" type="processor_e" />
340           </xs:sequence>
341         </xs:complexType>
342       </xs:element>
343       <xs:element name="Devices" type="devices_e"/>
344     </xs:sequence>
345   </xs:complexType>
346   <!-- Processor -->
347   <xs:complexType name="processor_e">
348     <xs:all>
349       <xs:element name="CyclicPlanTable" type="cyclicPlan_e"/>
350     </xs:all>
351     <xs:attribute name="id" type="id_t" use="required"/>
```

```
352     <xs:attribute name="frequency" type="freqUnit_t" use="optional" />
353     <xs:attribute name="features" type="processorFeaturesList_t" use="optional"
            default="none"/>
354     <xs:attribute name="console" type="idString_t" use="optional" />
355   </xs:complexType>
356   <!-- HwResource -->
357   <xs:complexType name="hwResources_e">
358     <xs:all>
359       <xs:element name="IoPorts" type="ioPorts_e" minOccurs="0" />
360       <xs:element name="Interrupts" minOccurs="0">
361         <xs:complexType>
362           <xs:attribute name="lines" type="hwIrqIdList_t" use="required"/>
363         </xs:complexType>
364       </xs:element>
365     </xs:all>
366   </xs:complexType>
367   <!-- Io Ports -->
368   <xs:complexType name="ioPorts_e">
369     <xs:sequence minOccurs="0" maxOccurs="unbounded">
370       <xs:choice>
371         <xs:element name="Range">
372           <xs:complexType>
373             <xs:attribute name="base" type="hex_t" use="required"/>
374             <xs:attribute name="noPorts" type="xs:positiveInteger" use="required
                    "/>
375           </xs:complexType>
376         </xs:element>
377         <xs:element name="Restricted">
378           <xs:complexType>
379             <xs:attribute name="address" type="hex_t" use="required"/>
380             <xs:attribute name="mask" type="hex_t" use="optional" default="0x0"/
                    >
381           </xs:complexType>
382         </xs:element>
383       </xs:choice>
384     </xs:sequence>
385   </xs:complexType>
386   <!-- CyclicPlan -->
387   <xs:complexType name="cyclicPlan_e">
388     <xs:sequence minOccurs="1" maxOccurs="unbounded">
389       <xs:element name="Plan" type="plan_e" />
390     </xs:sequence>
391   </xs:complexType>
392   <!-- Plan -->
393   <xs:complexType name="plan_e">
394     <xs:sequence minOccurs="1" maxOccurs="unbounded">
395       <xs:element name="Slot">
396         <xs:complexType>
397           <xs:attribute name="id" type="id_t" use="required"/>
398           <xs:attribute name="start" type="timeUnit_t" use="required"/>
399           <xs:attribute name="duration" type="timeUnit_t" use="required"/>
400           <xs:attribute name="partitionId" type="id_t" use="required"/>
401           <xs:attribute name="vCpuId" type="id_t" use="optional" default="0"/>
402           <xs:attribute name="flags" type="slotFlagsList_t" use="optional"/>
```

```
403         </xs:complexType>
404       </xs:element>
405     </xs:sequence>
406     <xs:attribute name="name" type="idString_t" use="optional"/>
407     <xs:attribute name="id" type="id_t" use="required"/>
408     <xs:attribute name="majorFrame" type="timeUnit_t" use="required"/>
409   </xs:complexType>
410   <!-- Health Monitor -->
411   <xs:complexType name="healthMonitor_e">
412     <xs:sequence minOccurs="1" maxOccurs="unbounded">
413       <xs:element name="Event">
414         <xs:complexType>
415           <xs:attribute name="name" type="hmString_t" use="required"/>
416           <xs:attribute name="action" type="hmAction_t" use="required"/>
417           <xs:attribute name="log" type="yntf_t" use="required"/>
418         </xs:complexType>
419       </xs:element>
420     </xs:sequence>
421   </xs:complexType>
422   <!-- Memory Layout -->
423   <xs:complexType name="memoryLayout_e">
424     <xs:sequence minOccurs="1" maxOccurs="unbounded">
425       <xs:element name="Region">
426         <xs:complexType>
427           <xs:attribute name="type" type="memRegion_t" use="required"/>
428           <xs:attribute name="start" type="hex_t" use="required"/>
429           <xs:attribute name="size" type="sizeUnit_t" use="required"/>
430         </xs:complexType>
431       </xs:element>
432     </xs:sequence>
433   </xs:complexType>
434   <!-- Hypervisor Memory Area -->
435   <xs:complexType name="hypMemoryArea_e">
436     <xs:attribute name="size" type="sizeUnit_t" use="required"/>
437     <xs:attribute name="flags" type="memAreaFlagsList_t" use="optional"/>
438   </xs:complexType>
439   <!-- Memory Area -->
440   <xs:complexType name="memoryArea_e">
441     <xs:sequence minOccurs="1" maxOccurs="unbounded">
442       <xs:element name="Area">
443         <xs:complexType>
444           <xs:attribute name="name" type="idString_t" use="optional" />
445           <xs:attribute name="start" type="hex_t" use="required"/>
446           <xs:attribute name="size" type="sizeUnit_t" use="required"/>
447           <xs:attribute name="flags" type="memAreaFlagsList_t" use="optional"
                  default="none"/>
448           <xs:attribute name="mappedAt" type="hex_t" use="optional"/>
449         </xs:complexType>
450       </xs:element>
451     </xs:sequence>
452   </xs:complexType>
453   <!-- Root Element -->
454   <xs:element name="SystemDescription">
455     <xs:complexType>
```

```
456       <xs:all>
457         <xs:element name="HwDescription" type="hwDescription_e" />
458         <xs:element name="XMHypervisor" type="hypervisor_e"/>
459         <xs:element name="ResidentSw" type="rsw_e" minOccurs="0"/>
460         <xs:element name="PartitionTable">
461           <xs:complexType>
462             <xs:sequence maxOccurs="unbounded">
463               <xs:element name="Partition" type="partition_e" />
464             </xs:sequence>
465           </xs:complexType>
466         </xs:element>
467         <xs:element name="Channels" type="channels_e" minOccurs="0" />
468       </xs:all>
469       <xs:attribute name="version" type="version_t" use="required"/>
470       <xs:attribute name="name" type="idString_t" use="required"/>
471     </xs:complexType>
472   </xs:element>
473   <!-- End Root Element -->
474   <!-- Elements -->
475 </xs:schema>
```

Listing A.1: `xmc.xsd`

## A.2 Configuration file example

```
<SystemDescription xmlns="http://www.xtratum.org/xm-3.x" version="1.0.0" name="
    Example">
       <HwDescription>
         <MemoryLayout>
            <Region type="rom" start="0x0" size="4MB" />
            <Region type="stram" start="0x40000000" size="4MB"/>
            <Region type="sdram" start="0x60000000" size="1MB"/>
         </MemoryLayout>
         <ProcessorTable>
           <Processor id="0" frequency="50Mhz">
             <CyclicPlanTable>
               <Plan id="0" majorFrame="2ms">
                 <Slot id="0" start="0ms" duration="1ms" partitionId="0"/>
                 <Slot id="1" start="1ms" duration="1ms" partitionId="1"/>
               </Plan>
             </CyclicPlanTable>
           </Processor>
         </ProcessorTable>
         <Devices>
           <Uart id="0" baudRate="115200" name="Uart"/>
           <MemoryBlock name="MemDisk0" start="0x4000" size="256KB" />
           <!--
              <MemoryBlock name="MemDisk0" start="0x40100000" size="256KB"/>
              <MemoryBlock name="MemDisk1" start="0x40150000" size="256KB"/>
              <MemoryBlock name="MemDisk2" start="0x40200000" size="256KB"/>
              -->
         </Devices>
       </HwDescription>
       <XMHypervisor console="Uart">
         <PhysicalMemoryArea size="512KB"/>
```

```
    </XMHypervisor>

    <!-- <track id="hello-xml-sample">-->
    <PartitionTable>
      <Partition id="0" name="Partition1" flags="system" console="Uart">
        <PhysicalMemoryAreas>
          <Area start="0x40080000" size="512KB"/>
        </PhysicalMemoryAreas>
        <TemporalRequirements duration="500ms" period="500ms"/>
      </Partition>
      <Partition id="1" name="Partition2" flags="system" console="Uart">
        <PhysicalMemoryAreas>
          <Area start="0x40100000" size="512KB" flags=""/>
        </PhysicalMemoryAreas>
        <TemporalRequirements duration="500ms" period="500ms"/>
        <HwResources>
          <Interrupts lines="4"/>
        </HwResources>
      </Partition>
    </PartitionTable>
</SystemDescription>
```

Listing A.2: XML configuration file example

This page is intentionally left blank.

# GNU Free Documentation License

Version 1.2, November 2002



51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA



### 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. Secondarily, 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.

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.

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.

## 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.

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.

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.

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.

## 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.

## 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.

## ADDENDUM: How to use this License for your documents

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’’.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with ... Texts." line with this:

```
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.

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.

# Glossary of Terms and Acronyms

## Glossary

**covert channel** A covert channel is a type of computer security flaw that allows to transfer information objects between processes that are not supposed to be allowed to communicate by the computer security policy.

**customisation file** A user defined file which is loaded in the memory space of XtratuM or the partitions. It is used to pass runtime configuration data to the partitions. For example the configuration vector to XtratuM; or runtime parameters to a partition.

**error** An error is the part of the system state that may cause a subsequent failure. A failure occurs when an error reaches the service interface and alters the service.

**failure** A failure is an event that occurs when the delivered service deviates from the correct service.

**fault** A fault is the adjudged or hypothesised cause of an error.

**hypercall** The service (system call) provided by the hypervisor. The services provided are known as para-virtual services.

**hypervisor** The layer of software that, using the native hardware resources, provides one or more virtual machines (partitions).

**i/o port** Also knwon as peripheral port, it is a low level processor address connected to an external peripheral. Some processors map the I/O ports in a designated memory addresses, and is accessed as if it were RAM memory; while others use a special I/O space which requires special processor isntructions.

**native hardware** The existing hardware: processor, interrupts, clock, etc.

**para-virtual** A virtual object that resembles, but with a different interface, the native object.

**partition** Also known as “virtual machine” or “domain”. It refers to the environment created by the hypervisor to execute user code.

**partition code** Also known as “guest”. It is the code executed inside a partition. Usually, the code is composed of an operating system and a set of processes or threads. Since application code relies on the services provided by the OS, we will assume that the partition code is an operating system (or a real-time operating system).

**resident software** The booting software that is executed directly in ROM memory right after a system reboot, also referred to as boot-loader or firmware. Among other tasks, it is in charge of loading XtratuM and the initial partitions in RAM memory.

**side channel** A side-channel is any observable information emitted as a byproduct of the physical implementation or operation of the system.

**slot** A conventionally defined time interval in a schedule.

**spare time** Processor time reserved for future utilisation. Note that idle time is the remaining processor capacity after the current workload has been fully attended.

**system partition** A partition that has extra capabilities to manage and to control the system and other partitions. Originally these partitions were named "supervisor partitions" but to avoid confusion with the processor modes they were renamed as "system partitions".

## Abbreviated terms

| Term | Description |
|---|---|
| ABI | Application Binary Interface. |
| APEX | APplication EXecutive. |
| API | Application Programming Interface. |
| ARINC | Aeronautical Radio, INC. http://www.arinc.com/ |
| BIOS | Basic Input Output Software. |
| bps | Bits Per Second. |
| CC | Common Criteria for Information Technology Security Evaluation. |
| ELF | Executable and Linkable Format. |
| ESD | Effective Slot Duration. |
| FIFO | First In First Out. |
| GPOS | General Purpose Operating System. |
| HM | Health Monitor. |
| IMA | Integrated Modular Avionics. |
| IPC | Inter Partition Communication. |
| MAF | Major Frame. See cyclic scheduling. |
| MMU | Memory Management Unit. |
| PCT | Partition Control Table. |
| PIT | Partition Information Table. |
| RSW | Resident SoftWare. |
| RTEMS | Real-Time Executive for Multiprocessor Systems. |
| SD | Slot Duration. |
| ST | Security Target. |
| TBR | Trap Base Register. A special LEON2 register. |
| TSC | TSF Scope of Control. |
| TSO | Total Storage Ordering. |
| UART | Universal Asynchronous Receiver Transmitter. A serial port. |
| VMM | Virtual Machine Monitor (hypervisor). |
| WCET | Worst Case Execution Time. |
| WIM | Window Invalid Mask. A special LEON2 register. |
| XAL | XtratuM Abstraction Layer. |
| XEF | XtratuM Executable Format. |
| XM_CF | XML XtratuM configuration file. It can also be named as `XM_CF.xml` to remind that it is an XML file. |
| XM_CT.bin | The compiled binary version of the `XM_CF` configuration file. |
| XML | eXtended Markup Languaje. |