XtratuM Hypervisor for x86

XtratuM User Manual

Fent Innovative Software Solutions

July, 2013
Reference: fnts-xm-um-41b

This page is intentionally left blank.

DOCUMENT CONTROL PAGE

TITLE: XtratuM Hypervisor for x86: XtratuM User Manual

AUTHOR/S: Fent Innovative Software Solutions

LAST PAGE NUMBER: 100

VERSION OF SOURCE CODE: XtratuM 2 for x86()

REFERENCE ID: fnts-xm-um-41b

SUMMARY: This guide describes the fundamental concepts of the XtratuM hypervisor

DISCLAIMER: This documentation is currently under active development. Therefore, no explicit or implied warranties in respect of any properties, including, but not limited to, correctness and fitness for purpose. Contributions of material, suggestions and corrections are welcome.

REFERENCING THIS DOCUMENT:

```
@techreport {fnts-xm-um-41b,
       title = {XtratuM Hypervisor for x86:  XtratuM User Manual},
       author = { Fent Innovative Software Solutions},
       institution = {Fent Innovative Software Solutions},
       number = {fnts-xm-um-41b},
       year={July, 2013},
}
```

Copyright © July, 2013 Fent Innovative Software Solutions

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

Changes:

Version	Date	Comments
0.1	February, 2013	[fnts-xm-um-41b] Initial document in the frame of the VOS4ES project
1.0	July, 2013	[fnts-xm-um-41b] Final document in the frame of the VOS4ES project

This page is intentionally left blank.

Contents

Preface

The audience for this document is software developers that have to use directly the services of XtratuM. The reader is expected to have strong knowledge of the x86 architecture and experience in programming device drivers. It is also advisable to have some knowledge of the ARINC-653 and related standards.

Typographical conventions

The following font conventions are used in this document:

- `typewriter`: used in assembly 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 like this:

```
static inline xmWord_t SaveStack(void) {
   xmWord_t sp;
   __asm__ __volatile__ ("movl %%esp, %0\n\t" : "=r" (sp) :: "memory");
   return sp;
}
```

Listing 1: Sample code

Caution sign

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

This page is intentionally left blank.

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 has 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 a relatively new. There have been some attempts, in the desktop and server markets, to standardise "how" an 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.1 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 an complete APEX.

In a bare 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 by 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.

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

Chapter 6 deals with the concrete structure and internal 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.

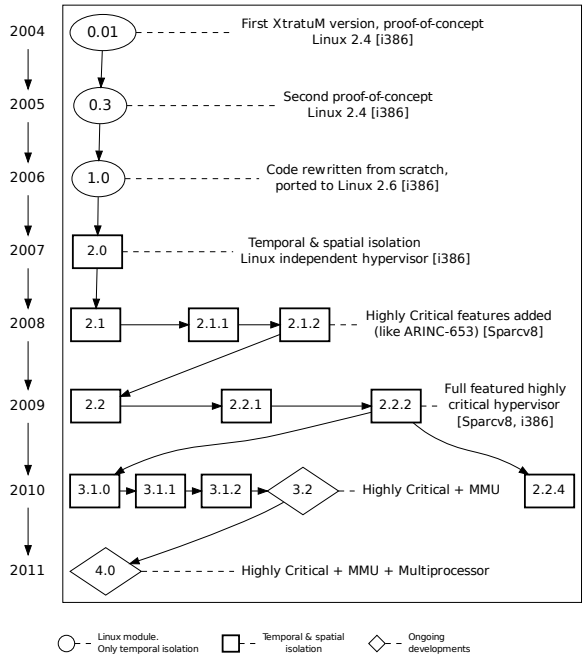


Figure 1.1: XtratuM evolution.

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 a truly 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 processor user mode, 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.

The current development version is 2.5, which has been derived from the LEON2 version and adapted for x86. Version 2.5 is stil under active development. The first stable version will be named 2.6. Hereinafter, the name XtratuM will be used to refer to the version 2.5 and eventually to 2.6 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 that allowed the Aeronautic Industry to manage the increment of the functionalities of the software maintaining the level of efficiency.

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

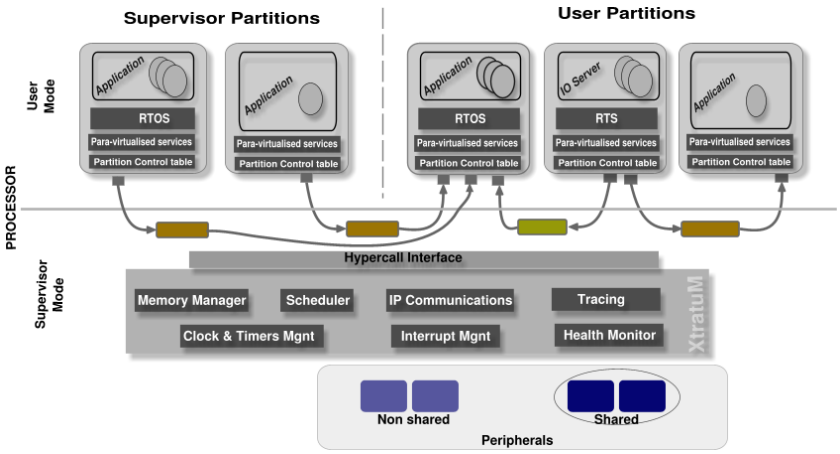


Figure 2.1: XtratuM architecture.

The main components of this architecture are:

- Hypervisor: XtratuM provides virtualisation services to partitions. It is executed in supervisor processor 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 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 a 32 additional interrupt numbers.

  - Clock and timer management.
  - IP communication: Inter-partition communication is related with 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 (implemented by the operating system of each partition), that share access to processor resources based upon the requirements of the application. The partition code can be: an application compiled to be executed on a bare-machine; a real-time operating system (or runtime support) and its applications; or a general purpose operating system and its applications.

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

Bare application : The application has to be virtualised using the services provided by XtratuM. The application is designed to run directly on the hardware and the hardware must be aware of this fact.

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. However, the operating system has to deal with the virtualisation and be virtualised (ported on top of XtratuM).

2.1 System operation

The system 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 the 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.

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

The system can switch to **halt** state by the health monitoring system in response to a detected error or by a *system partition* invoking the service `XM_halt_system()`. In the halt state: the scheduler is

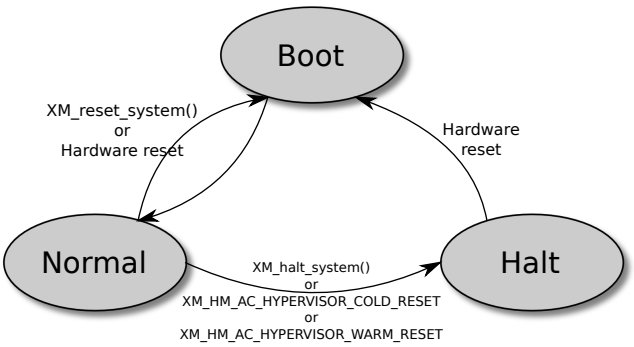


Figure 2.2: System states and transitions.

disabled, the hardware interrupts are disabled, and the processor enters in an endless loop. The only way to exit from this state is via an external hardware reset.

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

2.2 Partition operation

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

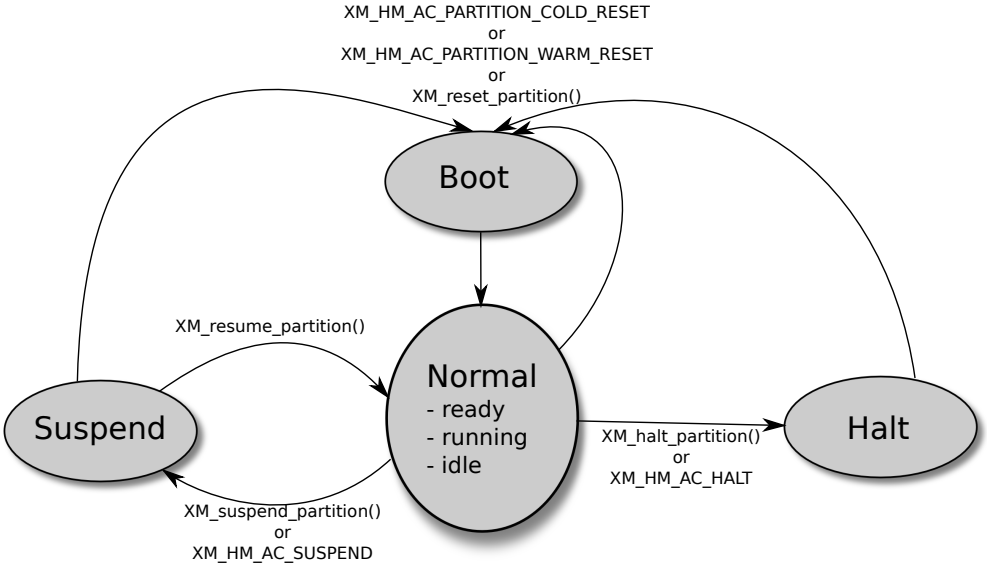


Figure 2.3: Partition states and transitions.

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 operating system has been initialised, the partition changes to normal mode.

The partition receives information from XtratuM about the previous executions, if any (see section 6.4.2).

From the point of view of the hypervisor, 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 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 waits for an interrupt or for the next time slot (see `XM_idle_self()`).

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

In suspend 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 the normal 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.

Note that system 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 an unique number *id*. Partition identifiers are assigned by the integrator in the `XM_CF` file. XtratuM uses this number 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.

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

Hypercall	System
XM_get_partition_status	Yes
XM_get_plan_status	Yes
XM_halt_partition	Partial
XM_halt_system	Yes
XM_hm_open	Yes
XM_hm_read	Yes
XM_hm_seek	Yes
XM_hm_status	Yes
XM_memory_copy	Partial
XM_reset_partition	Partial
XM_reset_system	Yes
XM_resume_partition	Yes
XM_set_plan	Yes
XM_set_spare_guest	Yes
XM_shutdown_partition	Partial
XM_suspend_partition	Partial
XM_system_get_status	Yes
XM_trace_open	Yes
XM_trace_read	Yes
XM_trace_seek	Yes
XM_trace_status	Yes

Table 2.1: List of system reserved hypercalls.

These id's 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 ordering 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 public symbols of XtratuM contain the prefix "xm". Therefore, the prefix "xm", both in upper and lower case, is reserved.

2.5 Partition scheduling

XtratuM schedules partitions in a fixed, cyclic basis (ARINC-653 scheduling policy). This policy ensures that one partition cannot utilise the processor for longer than intended to the detriment of the other partitions. The set of *time slots* allocated to each partition are defined in the XM_CF configuration during the design phase. Each partition is scheduled for a time slot defined as a starting 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 in 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).

[2]For efficiency and simplicity reasons.

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

(a) Partition set.

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

(b) Detailed execution plan.

Table 2.2: Partition definition.

For instance, consider the partition set of figure 2.2a, its hyper-period is 200 time units (milliseconds) and has a CPU utilisation of the 90%. The execution chronogram is depicted in figure 2.4. One of the possible cyclic scheduling plan 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.

```
<Processor id="0">
  <CyclicPlanTable>
    <Plan id="0" majorFrame="200ms">
       <Slot id="0" start="0ms" duration="20ms" partitionId="0 />
       <Slot id="1" start="20ms" duration="10ms" partitionId=1/>
       <Slot id="2" start="30ms" duration="10ms" partitionId="3 />
       <Slot id="3" start="40ms" duration="30ms" partitionId=2 />
       <Slot id="4" start="70ms" duration="10ms" partitionId="3 />

       <Slot id="5" start="100ms" duration="20ms" partitionId="0 />
       <Slot id="6" start="120ms" duration="10ms" partitionId=1/>
       <Slot id="7" start="130ms" duration="10ms" partitionId="3 />
       <Slot id="8" start="140ms" duration="30ms" partitionId=2 />
       <Slot id="9" start="170ms" duration="10ms" partitionId=3 />
       <Slot id="10" start="180ms" duration="20ms" partitionId=4/>
    </Plan>
  </CyclicPlanTable>
</Processor>
</Processor>
```

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 starting 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.

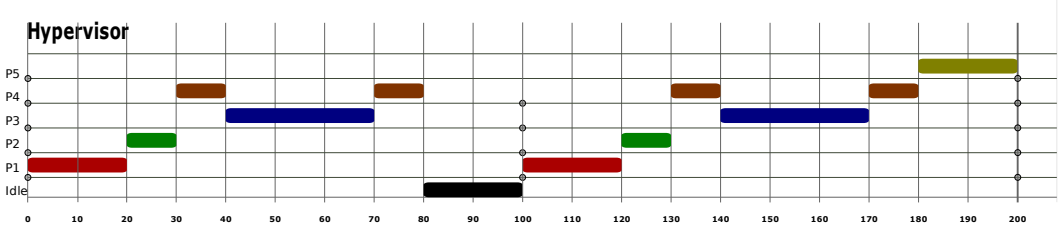


Figure 2.4: Scheduling example.

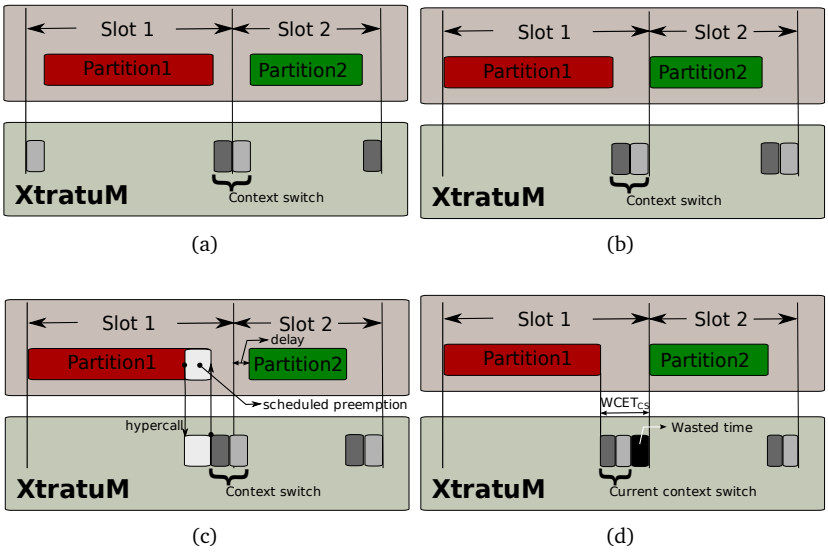


Figure 2.5: Xtratum context switch analysis.

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 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 ($WCET_{CS}$ - $BCET_{CC}$) + $WCET_{HC}$.

The cost of the context switch (both: $WCET_{CS}$ and $BCET_{CS}$) and all hypercalls have been evaluated and identified the worst case situation. 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 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 other are still initialising its data.
- The system can require to execute some maintenance operations. These operation can require other resource allocation than the operational mode.

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

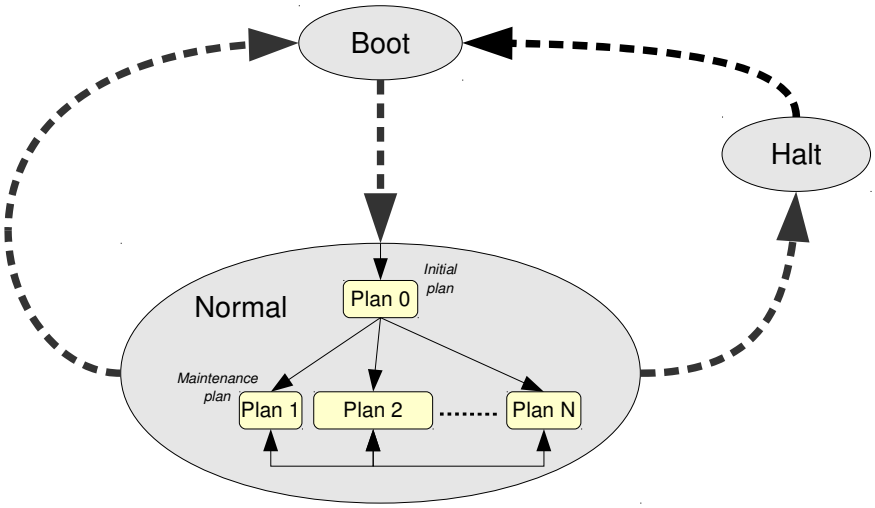


Figure 2.6: Scheduling modes.

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

Plan 0: *Initial plan*. The system executes 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 the a result of the health monitoring action XM_HM_AC_SWITCH_TO_MAINTENANCE. The plan switch is done immediately.
- 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 has 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 used defined. A system partition can switch to any 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 synchronous; 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.

2.6 Inter-partition communications (IPC)

Inter-partition communications are related with the 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 partition source to one or more partitions' destinations. 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 has 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, endianness, padding, etc.).

Channels, ports, maximum message sizes and maximum number of messages (queuing ports) are entirely defined in the configuration files (see section 7).

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

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

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

[3] XtratuM defines the maximum length of a message.

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).

2.7 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 when the `malloc()` function returns a null pointer when there are not enough memory to attend the request. This error is typically handled by the program by checking the return value. An attempt to execute an undefined instruction (processor instruction) may not be properly handled by the 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 the fault or confine the 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 a non-expected behaviour of the system, it may be difficult to clearly determine which is the original cause of the fault, and so, which is the best way to handle the problem. XtratuM provides a set of "coarse grain" actions (see section 2.7.2) that can be employed 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 two different execution scopes, depending on which part of the system has been initially affected:

1. Partition scope: Partition operating system or run-time support.
2. Hypervisor scope: XtratuM code.

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

2.7.1 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. **Health monitoring events raised by partitions are a special type of tracing message** (see sections 2.10). Highly critical tracing messages are considered as HM events.

[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

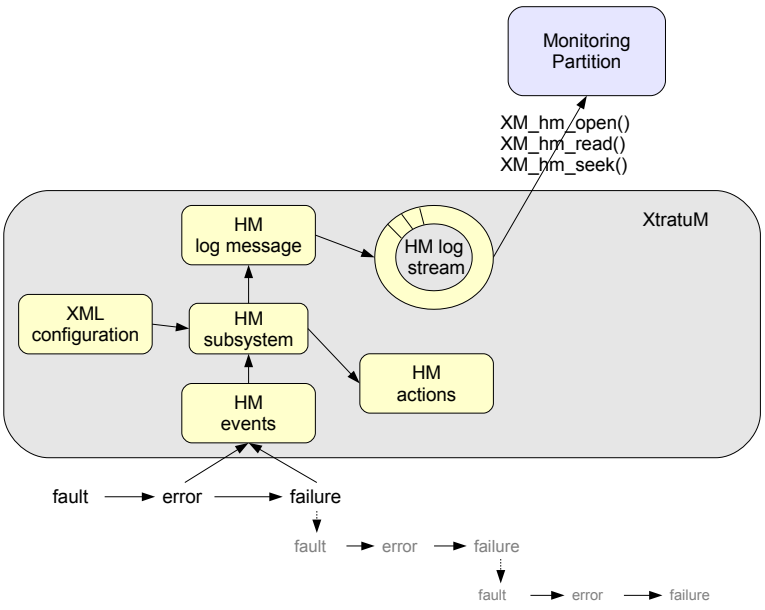


Figure 2.7: Health monitoring overview.

- 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 the 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.

2.7.2 HM Actions

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

Action	Description
XM_HM_AC_IGNORE	No action is performed.
XM_HM_AC_PARTITION_SHUTDOWN	The shutdown extended interrupt is sent to the failing partition.
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_PARTITION_SUSPEND	The failing partition is suspended.
XM_HM_AC_PARTITION_HALT	The failing partition is halted.
XM_HM_AC_SYSTEM_COLD_RESET	The failing processor is cold reset.
XM_HM_AC_SYSTEM_WARM_RESET	The failing processor is warm reset.
XM_HM_AC_SYSTEM_HALT	The failing processor is halted.
XM_HM_AC_PROPAGATE	No action is performed by XtratuM. The event is redirected to the partition as a virtual trap.

2.7.3 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 has to be managed at system or hypervisor scope.

Partition HM table: which defined the actions for those events that has to 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 an HM message after the execution of the HM action. It is possible to select whether an HM event is logged or not. See the chapter 7.

2.7.4 HM notification

The log events generated by the HM system (those event 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 using the `XM_hm_...` services.

The maximum number of messages on is configured in the XtratuM source code (see the section 7.1).

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

```
typedef struct {
    xm_u32_t eventId:13, system:1, reserved:2, moduleId:8, partitionId:8;
    union {
#define XM_HMLOG_PAYLOAD_LENGTH 5
        struct hmCpuCtxt cpuCtxt;
        xm_u32_t word[XM_HMLOG_PAYLOAD_LENGTH];
    };
    xmTime_t timeStamp;
} xmHmLog_t;
```

Listing 2.1: /core/include/objects/hm.h

`eventId`: Identifies the event that caused this log.

`system`: Set if the error was raised while executing XtratuM code.

`moduleId`: In the case of events raised by a partition (as a consequence of a high critical trace message), this field is a copy of the field with the same name of the trace message.

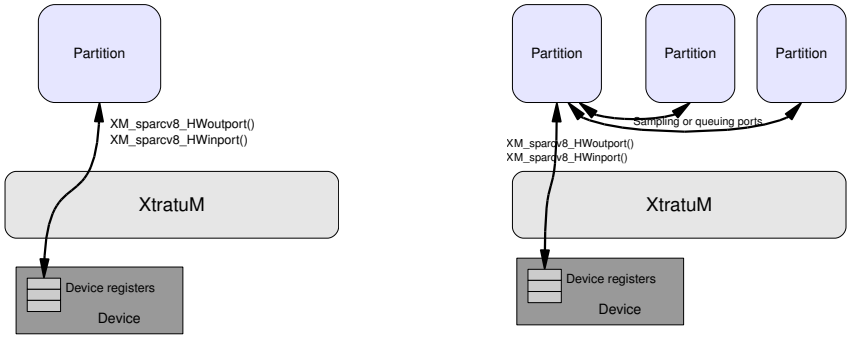
`partitionId`: The Id attribute of the partition that may caused the event.

`word`: Event specific information.

`timeStamp`: A time stamp of when the event was detected.

2.8 Access to devices

A partition, using exclusively a device (peripheral), can access the device through the device driver implemented in the partition (figure 2.8a). 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.



(a) Partition using exclusively a device.

(b) I/O Server partition.

Two partitions cannot use the same 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 a I/O port (see section 5.10).

When a device is used by several partitions, a user implemented I/O server partition (figure 2.8b) may be in charge of the device management. An I/O server partition is a specific partition which accesses and controls the devices attached to it, and exports a set of services via the inter-partitions 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.

2.9 Traps, interrupts and exceptions

2.9.1 Traps

A **trap** is the mechanism provided by the x86 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.

x86 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 `$idtr`). Both, the `$idtr` 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.

XtratuM defines 32 new interrupts called *extended interrupts*. These new interrupts are used to inform the partition about XtratuM specific events. Those new trap handlers are appended at the end of the native trap table.

Partitions are not allowed to use (read or write) the `$idtr` register. XtratuM implements a *virtual trap table*.

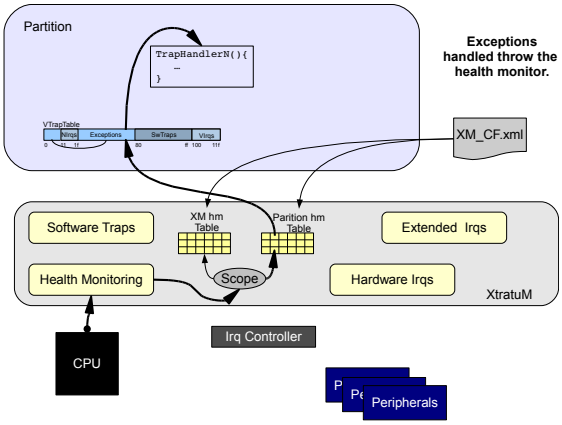


Figure 2.8: Exceptions handled by the health monitoring subsystem.

2.9.2 Interrupts

Although in a fully virtualised environment, a partition should not need to manage hardware interrupt; 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 the native interrupt controller using the `XM_mask_irq()` and `XM_unmask_irq()` functions.

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

Figure 2.9 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. The native trap table of the x86 is extended, appending 32 new trap entries, which will be invoked by XtratuM on the occurrence of an event alike a standard x86 trap.

2.10 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, which is specified in the `@device` attribute of the `Trace` element. `Trace` is an optional element of `XMHypervisor` and `Partition` elements.

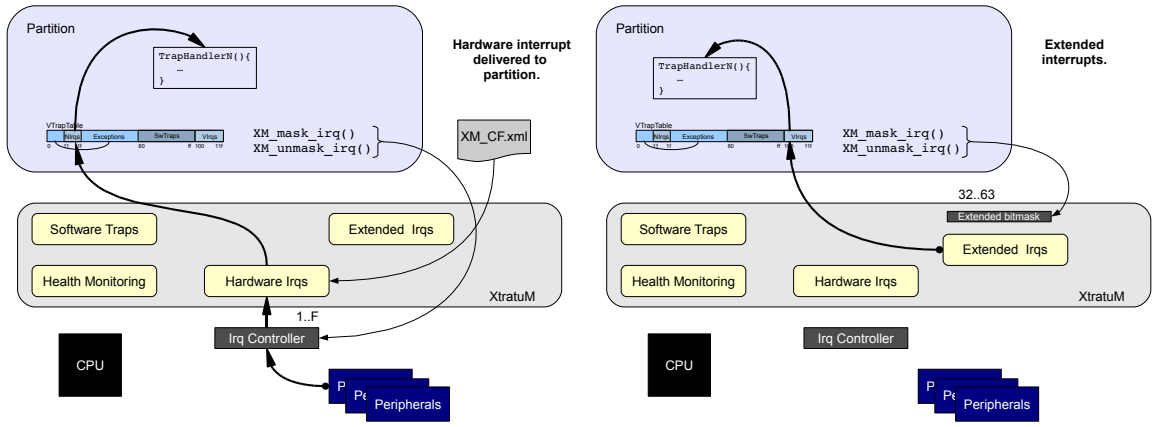


Figure 2.9: Hardware and extended interrupts delivery.

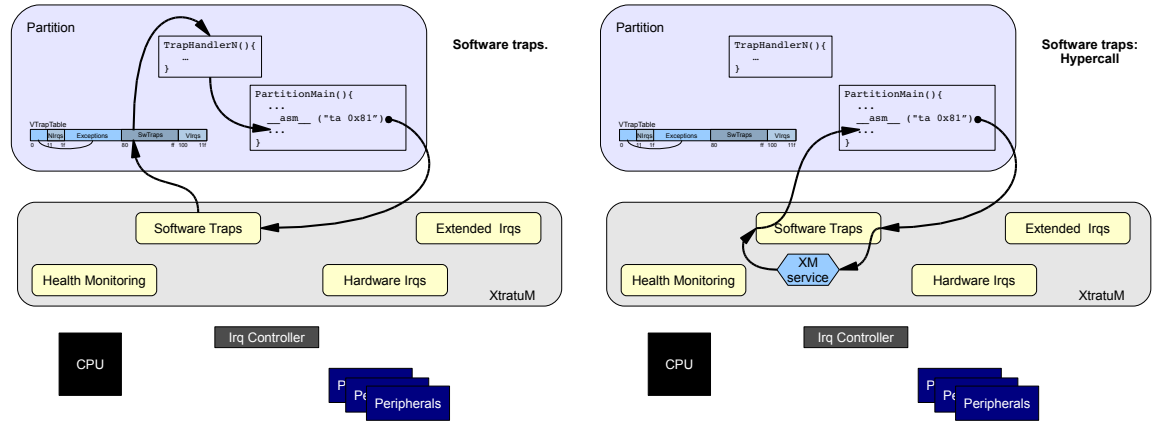


Figure 2.10: Software traps.

The hypercall to write a trace message has a parameter (bitmask) used to select the traces messages are stored in the log stream. The integrator can select which trace messages are actually stored in the log stream with the `Trace/@bitmask` attribute. If the logical and between the value configured in the `Partition/Trace/@bitmask` and the value of the `bitmask` parameter of the `XM_trace_event()` hypercall, then the event is stored, otherwise it is discarded.

Figure 2.11 sketches the configuration of the traces. In the example, the traces generated by partition 1 will be stored in the device `MemDisk0`, which is defined in the `Devices` section as a memory block device. Only those traces whose least significant bit is set in the bitmask parameter will be recorded.

2.11 Clocks and timers

There are two clocks per partition:

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

`XM_EXEC_CLOCK`: Associated with the execution of the partition. This clock only advances while the partition is being executed. It can be used by the partition to detect overruns. This clock relies on

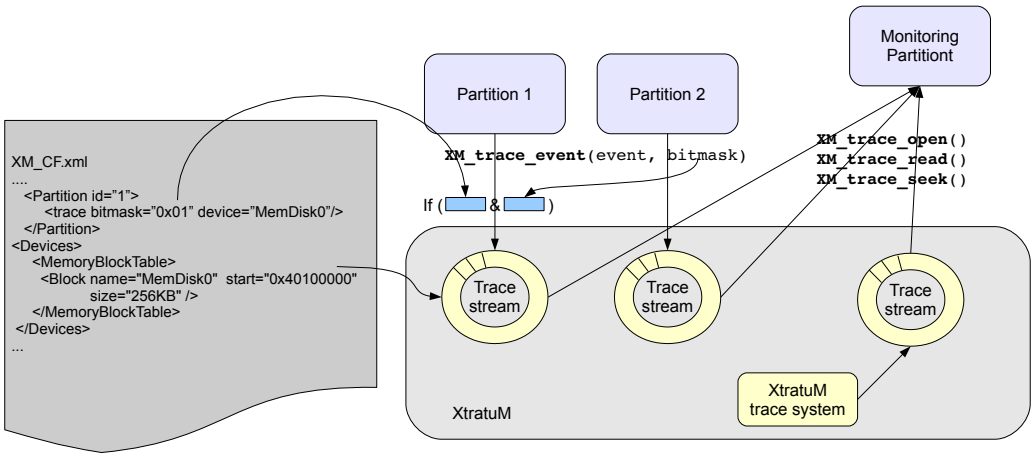


Figure 2.11: Tracing overview.

the `XM_HW_CLOCK` and its resolution is also 1μsec.

Only one timer can be armed for each clock.

2.12 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`.

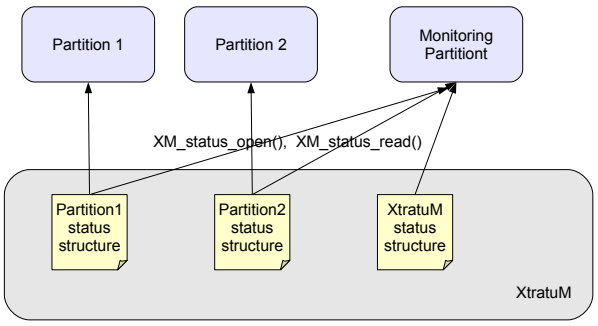


Figure 2.12: Status overview.

2.13 Summary

Next a is 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 should not be self-modifying.
- Partition's code is always executed with native interrupts enabled.
- Partition's code is not allowed to disable native interrupts, only 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 kind of virtual communication devices: sampling ports and queuing ports.
- All hypercall services are non-blocking.
- Regarding the capabilities of the partitions, XtratuM defines two kinds of partitions: system and standard.
- Only system partitions are allowed to control 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.

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 an hypervisor based one. This chapter provides an overview of the XtratuM developing environment.

The simplest scenario is composed of two actors: the *integrator* and a *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 7.1 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 7.3 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. Define the resources required by its application, and send it to the integrator.

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

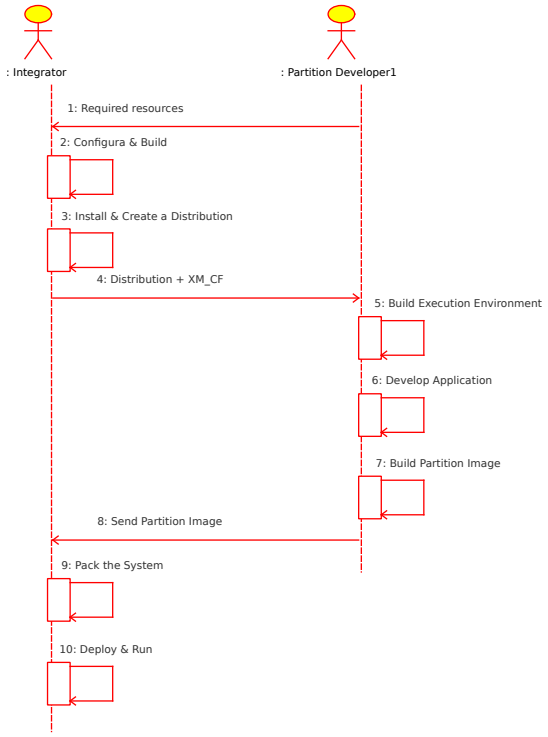


Figure 3.1: Integrator and partition developer interactions.

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.

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

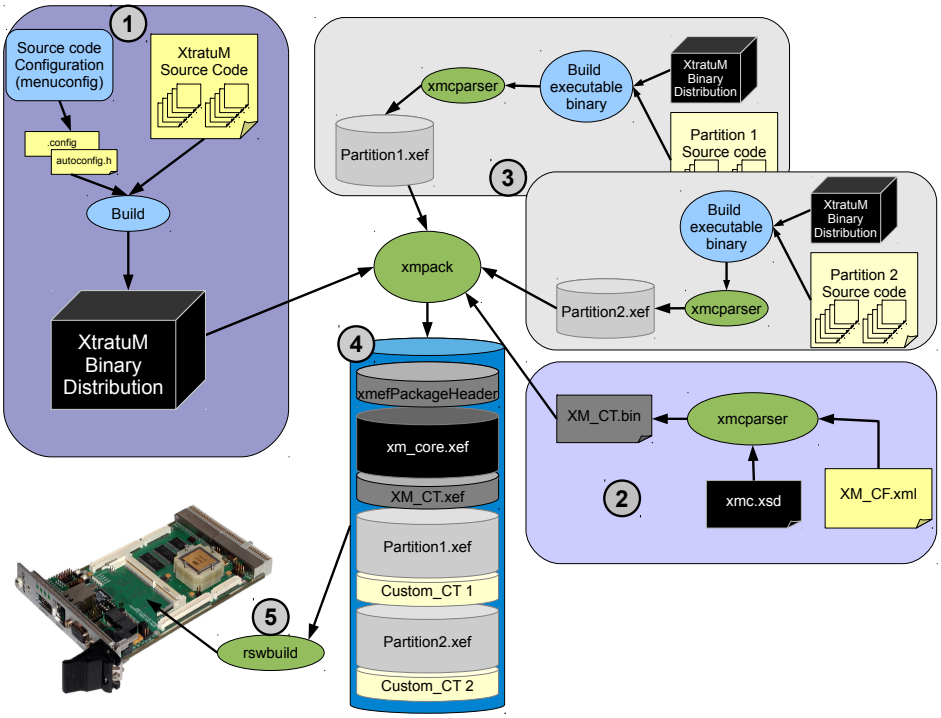


Figure 3.2: The big picture of building a XtratuM system.

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 three, 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 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 7.1).

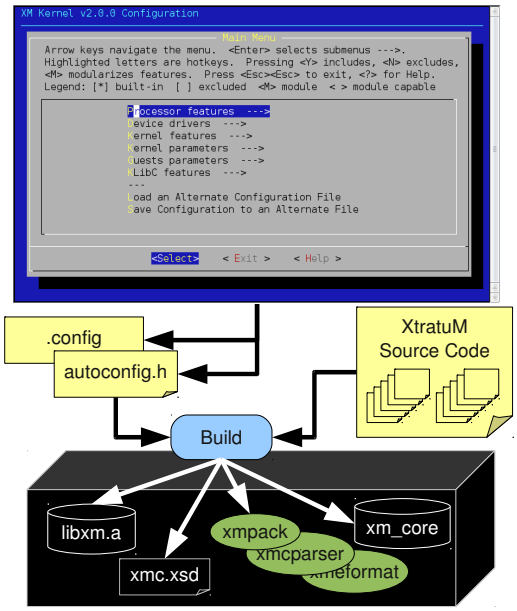


Figure 3.3: Menuconfig process.

The configuration interface is the same than 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 used in "C" programs and in Makefiles respectively.

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

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

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

The build process leaves the objects and executables files in the sources 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

The integrator, jointly with the partition developers, has 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 7.3. Figure 3.4 shows a graphical view of the configuration schema.

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

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

[2]TGZ: Tar GZipped archive.

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.

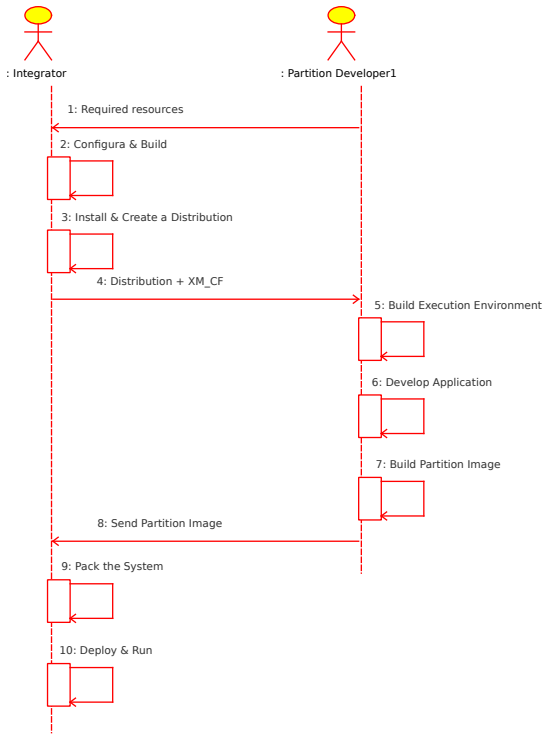


Figure 3.4: Graphical representation of an XML configuration file.

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 will need to contain an XML parser to read the `XM_CF` information. See section 8.1.1.

The resulting configuration binary will be 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 should 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 than the one used by the integrator. But the code of other partitions may be replaced by dummy partitions. This dummy partition code executes 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 three customisation files for 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. See section 6.

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

3.6 Building the final system image

In order to ensure that each partition does not depend on, or affects other partitions or the hypervisor, due to shared symbols. 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 will be 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 a bootable ELF file with the resident software and the container.

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	4:4.4.5-1	gcc-4.4	req	Build host utilities
make	3.81-8	make	req	Core
libncurses	5.7+20100313-5	libncurses5-dev	req	Configure source code
binutils	2.20.1-15	binutils	req	Core
x86-toolchain	4:4.4.5-1	gcc-4.4	req	Core
libxml2	2.7.8.dfsg-2	libxml2-dev	req	Configuration parser
qemu	0.11.1-1		opt	Simulated run
vmware	3.1.3 build-324285		opt	Simulated run
grub	1.98+20100804-11	grub2	opt	Deploy and run
perl	5.10.1-17	perl	opt	Testing
makeself	2.1.5	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. Make a deep clean to be sure that there is not previous configurations:

   ```
   $ make distclean
   ```

2. In the root directory of XtratuM, copy the file `xmconfig.x86` into `xmconfig`, and edit it to meet your system paths. The variable `XTRATUM_PATH` shall contain the root directory of XtratuM. Also, if the `gcc` toolchain directory is not in the `PATH` then the variable `TARGET_CCPREFIX` shall contain the path to the actual location of the corresponding tools.

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

```
$ cp xmconfig.ia32 xmconfig
```

3. 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 7.1.

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

4. Compile XtratuM sources:

```
$ make
> Configuring and building the "XtratuM hypervisor"
> Building XM Core
  - kernel/ia32
  - kernel/mmu
  - kernel
  - klibc
  - klibc/ia32
  - objects
  - devices
> Linking XM Core
   text    data     bss     dec     hex filename
  93404    8380   97636  199420   30afc xm_core
> Done

> Configuring and building the "User utilities"
> Building XM user
  - libxm
  - tools
  - tools/xmpack
  - tools/xmcparser
  - tools/rswbuild
  - tools/build_xmc
  - tools/elf2xef
  - bootloaders/rsw
  - xal
> Done
```

4.3 Generating binary a 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/xtratum-2.5.1-28129/xtratum-2.5.1/xm"
   - Generating XM sha1sums
   - Installing XAL
   - Generating XAL sha1sums
   - Setting read-only (og-w) permission.

> Generating XM distribution "xtratum-2.5.1.tar.bz2"
> Done

> Generating self extracting binary distribution "xtratum-2.5.1.run"
> Done
```

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

```
/install_path
├── xal
│   ├── bin
│   ├── common
│   ├── include
│   │   └── arch
│   └── lib
├── xal-examples
└── xm
    ├── bin
    ├── include
    │   ├── arch
    │   └── xm_inc
    └── user
        ├── bootloaders
        │   └── rsw
        └── tools
```

Figure 4.1: Content of the XtratuM distribution.

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 x86processor, 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 in 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:

710 1. The installation path.

2. The path to the x86compiler 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 x86toolchain 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
715 used to compile XtratuM.

```
$ ./xtratum-2.5.1.run
Verifying archive integrity... All good.
Uncompressing XtratuM binary distribution 2.5.1:
     ........................................
Starting installation.
Installation log in: /tmp/xtratum-installer-28821.log

Continue with the installation [Y/n]?

1.- Installation directory [/opt]: /opt/xm-sdk
2.- Path to the target toolchain [/usr/bin/]:

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

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 "Jordi-Fentiss" at "Tue Feb 12 13:26:09 CET
    2013"
XM path: "/opt/xm-sdk/xm"

XtratuM Core:
       Version: "2.5.1"
       Arch:    "ia32"
       File:    "/opt/xm-sdk/xm/user/xm_core.bin"
       Sha1:    "6b021ab855905e6808280745d59ef466e1fc28a4"
       Changed: ""

XtratuM Library:
       Version: "2.5.1"
       File:    "/opt/xm-sdk/xm/user/libxm.a"
       Sha1:    "6a97c7254af38c957d902dc32d029bd35ec8e289"
       Changed: ""

XtratuM Tools:
```

```
      File:   "/opt/xm-sdk/xm/bin/xmcparser"
      Sha1:   "b9345d4b87e14aaf4f264bc99751a09e092ec583"
      xmlns:  "http://www.xtratum.org/xm-2.3"
```

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`".

```
$ make resident_sw.iso
...
Enabling BIOS support ...
xorriso 1.2.4 : RockRidge filesystem manipulator, libburnia project.

$ qemu -m 512 -serial stdio -hda resident_sw.iso
...
>> HWClocks [TSC clock (2831686Khz)]
NOP idle selected
>> HwTimer [i8253 timer (1193Khz)]
[sched] using cyclic scheduler
1 Partition(s) created
P0 ("Partition1":0) cpu: 0 flags: [ SV BOOT (0x2000000) ]:
    [0x2000000 - 0x2100000]
Hello World!
[HYPERCALL] (0x0) Halted
```

This page is intentionally left blank.

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 error will be raised.

PIT and PCT: In the case of corrupting the Partition Control Table, the result on the faulting partition is undefined. The rest of the partitions are not affected.

Memory allocation:

- Care shall be taken to avoid overlapping the memory allocated to each partition.

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, an 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 that 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) : 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 in 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.ia32.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.xef: 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.

[1] para-virtualised operations provided by the hypervisor

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;

printf(‘‘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.

`stdio.c, stdio.h`: Minimal “C” support as `memcpy`, `printf`, etc.

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

The `boot.S` file:

```
 1
 2 #include <xm.h>
 3 #include <xm_inc/arch/asm_offsets.h>
 4
 5 #define NO_PGTS 8
 6 #define NO_IDT_ENTRIES 256
 7 #define NO_HWIRQS 16
 8 #define NO_EXTIRQS 32
 9
10 .align 4
11 .global __xmPartitionHdr
12 __xmPartitionHdr:
13         .long XMEF_PARTITION_HDR_MAGIC
14         .long start
15         .long __xmImageHdr
16         .long partitionControlTable
17         .long partitionInformationTable
18         .long _pgdAddr
19         .long (NO_PGTS+1)*4096
20
21 .text
22 .align 4
23
24 .global start, _start
25
26 _start:
27 start:
28         cld
29         xorl %eax,%eax
30         movl $_sbss,%edi
31         movl $_ebss,%ecx
32         subl %edi,%ecx
33         shrl $2,%ecx
34         rep ; stosl
35
36         mov $_estack, %esp
37
38         mov $write_register32_nr, %eax
39         mov $GDT_REG32, %ebx
40         mov $gdtDesc, %ecx
41         __XM_HC
42
43         ljmpl $((1<<3)|1), $1f
44 1:
45
46         mov $((2<<3)|1), %ebx
47         mov %ebx, %ds
48         mov %ebx, %ss
49         mov %ebx, %gs
50         mov %ebx, %fs
51
52         pushl (__xmPartitionHdr+
                _PARTITIONINFORMATIONTABLE_OFFSET)
53         pushl (__xmPartitionHdr+
```

```
            _PARTITIONCONTROLTABLE_OFFSET)
54          call init_libxm
55          addl $8, %esp
56          call InitArch
57
58          call PartitionMain
59
60          mov $halt_partition_nr, %eax
61          mov (partitionInformationTable+
                _ID_OFFSET), %ebx
62          __XM_HC
63  1:
64          jmp 1b
65
66  #define HW_SAVE_ALL \
67      cld ; \
68      pushl %gs ; \
69      pushl %fs ; \
70      pushl %es ; \
71      pushl %ds ; \
72      pushl %eax ; \
73      pushl %ebp ; \
74      pushl %edi ; \
75      pushl %esi ; \
76      pushl %edx ; \
77      pushl %ecx ; \
78      pushl %ebx
79
80  #define HW_RESTORE_ALL \
81      popl %ebx ; \
82      popl %ecx ; \
83      popl %edx ; \
84      popl %esi ; \
85      popl %edi ; \
86      popl %ebp ; \
87      popl %eax ; \
88      popl %ds ; \
89      popl %es ; \
90      popl %fs ; \
91      popl %gs ; \
92      addl $8, %esp
93
94  CommonTrapBody:
95          HW_SAVE_ALL
96          pushl %esp
97          call DoTrap
98          addl $4, %esp
99          HW_RESTORE_ALL
100         jmp XM_iret
101
102 .macro TABLE_START section, symbol
103 .section .rodata.\section\(),"a"
104 .globl \symbol\()
105 .align 4
106 \symbol\() :
107 .endm
108
109 .macro TABLE_END section
110 .section .rodata.\section\(),"a"
111 .align 4
112 .long 0
113 .previous
114 .endm
115
116 .macro BUILD_TRAP_ERRCODE trap
117 .section .rodata.trapHndl,"a"
118 .align 4
119 .long 1f
120 .text
121 .align 4
122 1:
123         pushl $\trap\() /* error_code has
                already been filled */
124         jmp CommonTrapBody
125 .endm
126
127 .macro BUILD_TRAP_NOERRCODE trap
128 .section .rodata.trapHndl,"a"
129 .align 4
130 .long 1f
131 .text
132 .align 4
133 1:
134         pushl $0 /* the error_code (no error
                code) */
135         pushl $\trap\() /* the trap */
136         jmp CommonTrapBody
137 .endm
138
139 .macro BUILD_TRAP_BLOCK stTrapNr endTrapNr
140 vector=\stTrapNr\()
141 .rept \endTrapNr\()-\stTrapNr\()
142         BUILD_TRAP_NOERRCODE vector
143 vector=vector+1
144 .endr
145 .endm
146
147 TABLE_START trapHndl, trapTable
148 BUILD_TRAP_NOERRCODE(0x0)
149 BUILD_TRAP_NOERRCODE(0x1)
150 BUILD_TRAP_NOERRCODE(0x2)
151 BUILD_TRAP_NOERRCODE(0x3)
152 BUILD_TRAP_NOERRCODE(0x4)
153 BUILD_TRAP_NOERRCODE(0x5)
154 BUILD_TRAP_NOERRCODE(0x6)
155 BUILD_TRAP_NOERRCODE(0x7)
156 BUILD_TRAP_ERRCODE(0x8)
157 BUILD_TRAP_NOERRCODE(0x9)
158 BUILD_TRAP_ERRCODE(0xa)
159 BUILD_TRAP_ERRCODE(0xb)
160 BUILD_TRAP_ERRCODE(0xc)
161 BUILD_TRAP_ERRCODE(0xd)
162 BUILD_TRAP_ERRCODE(0xe)
163 BUILD_TRAP_NOERRCODE(0xf)
164 BUILD_TRAP_NOERRCODE(0x10)
165 BUILD_TRAP_ERRCODE(0x11)
166 BUILD_TRAP_NOERRCODE(0x12)
167 BUILD_TRAP_NOERRCODE(0x13)
168 BUILD_TRAP_ERRCODE(0x14)
169 BUILD_TRAP_ERRCODE(0x15)
170 BUILD_TRAP_ERRCODE(0x16)
171 BUILD_TRAP_ERRCODE(0x17)
172 BUILD_TRAP_ERRCODE(0x18)
173 BUILD_TRAP_ERRCODE(0x19)
174 BUILD_TRAP_ERRCODE(0x1a)
175 BUILD_TRAP_ERRCODE(0x1b)
176 BUILD_TRAP_ERRCODE(0x1c)
177 BUILD_TRAP_ERRCODE(0x1d)
178 BUILD_TRAP_ERRCODE(0x1e)
179 BUILD_TRAP_ERRCODE(0x1f)
180 BUILD_TRAP_BLOCK 0x20 NO_IDT_ENTRIES
181 TABLE_END trapHndl
182
183 .data
184         .word 0
185 .global idtDesc
```

```
186 .align 4
187 idtDesc:
188         .word NO_IDT_ENTRIES*8-1
189         .long idtTab
190
191 gdtTab:
192         .quad 0x0000000000000000 /* NULL
                descriptor */
193         .quad 0x00cfba000000bfff /* 1<<3 code
                segment R1 */
194         .quad 0x00cfb2000000bfff /* 2<<3 data
                segment R1 */
195
196  .word 0
197 ENTRY(gdtDesc)
198         .word 3*8-1
199         .long gdtTab
200
201 .bss
202
203 _stack:
204         .zero STACK_SIZE
205 _estack:
206
207 .globl idtTab
208 idtTab:
209         .zero (NO_IDT_ENTRIES*8)
210
211 .section .xm_ctrl, "w"
212 .align 4096
213 _pgdAddr:
214         .zero 4096
215         .zero (4096*NO_PGTS)
216
217 .previous
```

Listing 5.6: /user/xal/ia32/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`:
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-i386", "elf32-i386
    ", "elf32-i386")
OUTPUT_ARCH("i386")
ENTRY(start)

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

  .rodata ALIGN (4) : {
        *(.rodata)
        *(.rodata.*)
        *(.rodata.*.*)
  }

  .data ALIGN (4) : {
        _sdata = .;
        *(.data)
        _edata = .;
  }

  .bss ALIGN (4) : {
        *(.xm_ctrl)
        _sbss = .;
        *(COMMON)
        *(.bss)
        _ebss = .;
  }

  _eguest = .;

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

Listing 5.7: /user/xal/ia32/loader.lds

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

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

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

Listing 5.8: Ported simple example

Listing 5.9 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 5.9: 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 0x800000. This address shall be the same than the one declared in the XM_CF

file:
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.ia32.xml /SystemDescription/PartitionTable/Partition
    [1]/PhysicalMemoryAreas/Area[1]/@start
0x800000
```

Listing 5.10: 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:

5.5 Partition reset

A partition reset is an unconditional jump to the partition entry point. 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()`).
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 7.3.5).

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. 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.

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

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 5.11: /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 5.12: 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/stdio.c`. Some of these functions are: `strlen()`, `printf()` which are similar to the functions provided by a `stdio`.
The use of `printf()` is illustrated in the next example:

```
#include <xm.h>
#include "stdio.h"     // header of the stdio.h

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

while(1) {
counter++;
```

```
if (!(counter%1000))
printf("%d\n", counter);
}
}
```

Listing 5.13: Ported dummy code 1

`printf()` 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 a 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. `XM_create_sampling_port()`

```
#include <xm.h>
#include "stdio.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 ) {
printf("[%s] cannot be created", PORT_NAME);
return;
}

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


}
```

Listing 5.14: Partition_1

```
#include <xm.h>
#include "stdio.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 ) {
printf("[%s] cannot be created", PORT_NAME);
return;
}

while(1) {
XM_read_sampling_message(portDesc,
&value,
sizeof(value),
&flags);
if (!(value == previous)){
printf("%d\n", value);
previous = value;
}
}
}
```

Listing 5.15: 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 Device Management

Device management in x86 is a very complex subject. Device management is not addressed by XtratuM for complex devices. Access to devices must be granted through the XML configuration file, where the relevant I/O ports, memory areas and interrupts must be configured. Devices may be accessed and configured through three different spaces:

IO address space: x86 provides a sepparate address space for memory management. This address space is composed of 65536 I/O ports, each of which is accessed through the special instructions *in* and *out*.

Memory mapped devices: Devices may also be memory mapped. In this case access shall be granted to the whole memory area where the device resides.

Indirect device configuration spaces: There are special cases as, for example, PCI based devices. PCI offers a PCI device configuration space that is indirectly accessed through two I/O ports, namely, 0xCF8 and 0xCFC. Hence, giving access to this two ports to a specific partition would also allow the partition to configure all the devices plugged in the PCI bus.

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.
Port access is restricted through the partition Task State Segment (TSS). The TSS contains a 65536 bits bitmap that allows a specific partition access the ports whose corresponding bits are cleared. If the partition tries to access a port with an *in/out* instruction to which it has no access rigths, the processor will raise a #GP exception. This way, the I/O address space can be isolated between 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 is the same than the native trap table. Each entry is 16 bytes and contains the trap handler routine (which in the practice, is branch or jump instruction to the real handler).
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:

`0..31`: 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).

`32..63`: Correspond to the XtratuM extended interrupts.

These interrupts are assigned to traps 0x100 to 0x1F.

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

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

#define XM_VT_EXT_FIRST        (32)
#define XM_VT_EXT_MAX          (32)
#define XM_VT_EXT_LAST         (XM_VT_EXT_FIRST+XM_VT_EXT_MAX
    -1)

#define XM_VT_EXT_HW_TIMER      (0+XM_VT_EXT_FIRST)
#define XM_VT_EXT_EXEC_TIMER    (1+XM_VT_EXT_FIRST)
#define XM_VT_EXT_WATCHDOG_TIMER (2+XM_VT_EXT_FIRST)
#define XM_VT_EXT_SHUTDOWN      (3+XM_VT_EXT_FIRST)
#define XM_VT_EXT_OBJDESC       (4+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)

/* <track id="xm-ipvi-list"> */
/* Inter-Partition Virtual Interrupts */
#define XM_MAX_IPVI 8

#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)
/* </track id="xm-ipvi-list"> */
```

Listing 5.16: /core/include/guest.h

All hardware and extended interrupt, can be masked through the following hypercalls: `XM_mask_irq()` and `XM_unmask_irq()`. Besides, all these set of interrupts can be globally disabled/enable by using the `XM_disable_irqs()` and `XM_enable_irqs()` respectively.

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 (register window over/underflow) but others are

caused by an abnormal situation (invalid instruction).
Error related exception traps, are managed by XtratuM thorough the health monitoring system.

```
#define IA32_DIVIDE_ERROR            (0x00)  // 0
#define IA32_RESERVED1_EXCEPTION     (0x01)  // 1
#define IA32_NMI_INTERRUPT           (0x02)  // 2
#define IA32_BREAKPOINT_EXCEPTION    (0x03)  // 3
#define IA32_OVERFLOW_EXCEPTION      (0x04)  // 4
#define IA32_BOUNDS_EXCEPTION        (0x05)  // 5
#define IA32_INVALID_OPCODE          (0x06)  // 6
#define IA32_COPROCESSOR_NOT_AVAILABLE (0x07)  // 7
#define IA32_DOUBLE_FAULT            (0x08)  // 8
#define IA32_COPROCESSOR_OVERRRUN    (0x09)  // 9
#define IA32_INVALID_TSS             (0x0a)  // 10
#define IA32_SEGMENT_NOT_PRESENT     (0x0b)  // 11
#define IA32_STACK_SEGMENT_FAULT     (0x0c)  // 12
#define IA32_GENERAL_PROTECTION_FAULT (0x0d)  // 13
#define IA32_PAGE_FAULT              (0x0e)  // 14
#define IA32_RESERVED2_EXCEPTION     (0x0f)  // 15
#define IA32_FLOATING_POINT_ERROR    (0x10)  // 16
#define IA32_ALIGNMENT_CHECK         (0x11)  // 17
#define IA32_MACHINE_CHECK           (0x12)  // 18
```

Listing 5.17: /core/include/ia32/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:

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

Listing 5.18: /core/include/hypercalls.h

XtratuM provides one timer for each clock. The timers can be programmed on 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 1μsec. 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 "stdio.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);
printf("Initial time: %lld, final time: %lld", t1, t2);
printf("Difference: %lld\n", t2-t1);
XM_halt_partition(XM_PARTITION_SELF);
}
```

5.13 Tracing

5.13.1 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 *opCode* and an associated user defined data:

```
typedef struct {
    xmTraceOpCode_t opCode;
    xm_u32_t reserved;
    xmTime_t timeStamp;
    union {
        xm_u32_t word[4];
        char str[16];
    };
} xmTraceEvent_t;
```

Listing 5.19: /core/include/objects/trace.h

The type xmTraceOpCode_t is a 32bit value with the following bit fields:

```
typedef struct {
    xm_u32_t code:13, criticality:3, moduleId:8, partitionId:8;
#define XM_TRACE_UNRECOVERABLE 0x3 // This level triggers a health
                                   // monitoring fault
#define XM_TRACE_WARNING 0x2
#define XM_TRACE_DEBUG 0x1
#define XM_TRACE_NOTIFY 0x0
} xmTraceOpCode_t;
```

Listing 5.20: /core/include/objects/trace.h

partitionId: Identify the partition who issued the trace event. This field is automatically filled by XtratuM. The value XM_HYPERVISOR_ID is used to identify XtratuM traces.

moduleId: For the traces issued by the partitions, this filed is user defined. For the traces issued by XtratuM, this field identifies an internal subsystem:

TRACE_MODULE_HYPERVISOR Traces related to XtratuM core events.

TRACE_MODULE_PARTITION Traces related to partition operation.

TRACE_MODULE_SCHED Traces concerning scheduling.

`code`: This filed is user defined for the traces issued by the partitions. For the traces issued by XtratuM, the values of the `code` field depends on the value of the `moduleId`:

If `moduleId` = `TRACE_MODULE_HYPERVISOR`

`TRACE_EV_HYP_HALT`: The hypervisor is about to halt.

`TRACE_EV_HYP_RESET`: The hypervisor is about to perform a software reset.

`TRACE_EV_HYP_AUDIT_INIT`: The first message after the audit startup.

If `moduleId` = `TRACE_MODULE_PARTITION`

The field `auditEvent.partitionId` has the identifier of the affected partition. The recorded events are:

`TRACE_EV_PART_SUSPEND`: The affected partition has been suspended.

`TRACE_EV_PART_RESUME`: The affected partition has been resumed.

`TRACE_EV_PART_HALT`: The affected partition has been halted.

`TRACE_EV_PART_SHUTDOWN`: A shutdown extended interrupt has been delivered to the partition.

`TRACE_EV_PART_IDLE`: The affected partition has been set in idle state.

`TRACE_EV_PART_RESET`: The affected partition has been reset.

If `moduleId` = `TRACE_MODULE_SCHED`

The field `auditEvent.partitionId` has the identifier of the partition that requested the plan switch; or `XM_HYPERVISOR_ID` if the plan switch is the consequence of the `XM_HM_AC_SWITCH_TO_MAINTENANCE` health monitoring action.

The field `auditEvent.newPlanId` has the identifier of the new plan.

`TRACE_EV_SCHED_CHANGE_REQ`: A plan switch has been requested.

`TRACE_EV_SCHED_CHANGE_COMP`: A plan switch has been carried out.

`criticality`: Determines the importance/criticality of the event that motivated the trace message. Next are the intended use of the levels:

`XM_TRACE_NOTIFY` A notification messages of the progress of the application at coarse-grained level.

`XM_TRACE_DEBUG` A detailed information message intended to be used for debugging during the development phase.

`XM_TRACE_WARNING` Traces which informs about potentially harmful situations.

`XM_TRACE_UNRECOVERABLE` Traces of this level are managed also by the health monitoring subsystem: a user HM event is generated, and handled according to the HM configuration. Note that both, the normal partition trace message is stored, and the HM event is generated.

Jointly with the `opCode` and the user data, the `XM_trace_event()` function has a bitmask parameter that is used to filter out trace events. If the logical `AND` between the bitmask parameter and the bitmask of the `XM_CF` configuration file is not zero then the trace event is logged; otherwise it is discarded. Traces of `XM_TRACE_UNRECOVERABLE` critically always raises a health monitoring event regarding the bitmask.

5.13.2 Reading traces

Only one system partition can read from a trace stream. A standard partition **can not read its own trace messages**, it is only allowed to store traces on it.
If the trace stream is stored in a buffer (RAM or FLASH). When the buffer is full, the oldest events are overwritten.

5.13.3 Configuration

XtratuM statically allocates a block of memory to store all traces. The amount of memory reserved to store traces is a configuration parameter of the sources (see section 7.1).
In order to be able to store the traces of a partition, as well as the traces generated by XtratuM, it has to be properly configured in the `XM_CF` configuration file. The `bitmask` attribute is used to filter which traces are stored. The traces recoded by XtratuM can be selected (masked) at module granularity.

5.14 System and partition status

The hypercalls `XM_get_partition_status()` and `XM_get_system_status()` return information about a given partition and the system respectively.
The data structure returned are:

```
typedef struct {
    /* Current state of the partition: ready, suspended ... */
    xm_u32_t state;
#define XM_STATUS_IDLE 0x0
#define XM_STATUS_READY 0x1
#define XM_STATUS_SUSPENDED 0x2
#define XM_STATUS_HALTED 0x3

    xm_u32_t opMode;
#define XM_OPMODE_IDLE 0x0
#define XM_OPMODE_COLD_RESET 0x1
#define XM_OPMODE_WARM_RESET 0x2
#define XM_OPMODE_NORMAL 0x3

    /* Number of virtual interrupts received. */
    xm_u64_t noVIrqs;               /* [[OPTIONAL]] */
    /* Reset information */
    xm_u32_t resetCounter;
    xm_u32_t resetStatus;
```

Listing 5.21: Partition status.

```
typedef struct {
    xm_u32_t resetCounter;
    /* Number of HM events emmite. */
    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]] */
} xmSystemStatus_t;
```

Listing 5.22: System status.

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

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

5.15 Memory management

x86memory manement is done in two levels. The first level of memory management is the physical to virtual mappings. This level allows the mapping of physical pages 4KB or 4MB to different virtual addresses. At this level, rough read/write protection, caching and page size management is controlled. The second level is the segmentation. Segments are controlled through the Global Descriptor Table. This table defines the protection details of each virtual memory region in a more accurate and precise way. Through this level, user/supervisor permissions, as well as fine grained memory access control, is controlled. Also, unlike paging, the segment sizes may vary freely.

5.15.1 Paging

Paging is the way by which x86 processors handle physical to virtual mappings of pages. Pages may be wither of 4 KB or of 4 MB. The paging is controlled via page tables. As usual, page tables conform a hierarchical construct, which is controlled by the x86 `%cr3` register. In turn, in order to access page tables, this must be mapped by the very mechanism of the page tables. Pages can be defined as read-only. This way, partitions are granted read-only access to the page tables. This allows for better partition performance, as reading the page tables does not need the hypervisor to intermediate, while keeps spatial isolation.

5.15.2 Segmentation

Segmentation is the way by which x86 processors handle privilege levels and the mapping of segments to the virtual address space, also known as the linear address space. The segmentation is controlled in x86 processors by the Global Descriptor Table. This table is completely controlled by XtratuM for all partitions. In turn, each partition is provided with relevant hypercalls to update a GDT entry, or the whole table at once. XtratuM will do the relevant checks, so that the partition cannot insert a GDT entry that may endanger isolation. In particular, this checks include:

- Check that the segment descriptor inserted has a descriptor privilege level greater or equal than 1.
- Check that the segment range does not exceed the CONFIG_XM_OFFSET limit, over which XtratuM is mapped.

5.16 Releasing the processor

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

5.17 Partition customisation files

A partition is composed of a binary image (code and data) and, zero or more additional files (customisation files). To ease the management of these additional files, the header of the partition image (see section 6.4.1) 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.

5.18 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 5.23) and assembly hypercalls (listing 5.24):

```
#define __MULTICALL_NR 0
#define __HALT_PARTITION_NR 1
#define __SUSPEND_PARTITION_NR 2
#define __RESUME_PARTITION_NR 3
#define __RESET_PARTITION_NR 4
#define __SHUTDOWN_PARTITION_NR 5
#define __HALT_SYSTEM_NR 6
#define __RESET_SYSTEM_NR 7
#define __IDLE_SELF_NR 8
#define __WRITE_REGISTER32_NR 9
#define __GET_TIME_NR 10
#define __SET_TIMER_NR 11
#define __READ_OBJECT_NR 12
#define __WRITE_OBJECT_NR 13
#define __SEEK_OBJECT_NR 14
#define __CTRL_OBJECT_NR 15
#define __MASK_HWIRQ_NR 16
#define __UNMASK_HWIRQ_NR 17
#define __UPDATE_PAGE32_NR 18
#define __SET_PAGE_TYPE_NR 19
#define __WRITE_REGISTER64_NR 20
#define __OVERRIDE_TRAP_HNDL_NR 21
#define __RAISE_IPVI_NR 22
#define ia32_update_sys_struct_nr 23
```

Listing 5.24: xm_inc/arch/hypercalls.h

The file "core/include/ia32/hypercalls.h" has additional services for the x86 architecture.

5.18.1 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_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 5.25: /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:

```
__stdcall xm_s32_t XM_hm_status(xmHmStatus_t *hmStatusPtr) {
    return XM_ctrl_object(OBJDESC_BUILD(OBJ_CLASS_HM, XM_HYPERVISOR_ID, 0), XM_HM_GET_STATUS,
        hmStatusPtr);
}
```

Listing 5.26: /user/libxm/common/hm.c

5.19 Manpages summary

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

Hypercall	Description
XM_are_irqs_enabled	Checks if interrupts are enabled.
XM_create_queuing_port	Create a queuing port.
XM_create_sampling_port	Create a sampling port.
XM_ctrl_object	Performs a control operation on a object.
XM_disable_irqs	Disable processor irqs.
XM_enable_irqs	Enable processor irqs.
XM_exec_pendirqs	Executes any pending hardware or extended interrupts.
XM_flush_hyp_batch	Flush a batch of multiple hypercalls.
XM_get_partition_status	Get the current status of a partition.
XM_get_physmem_map	Returns the physical memory map of the partition.
XM_get_plan_status	Return information about the scheduling plans.

Hypercall	Description
XM_get_queuing_port_status	Get the status of a queuing port.
XM_get_sampling_port_status	Get the status of a sampling port.
XM_get_time	Retrieve the time of the specified clock.
XM_halt_partition	Terminates a partition.
XM_halt_system	Stop the system.
XM_hm_open	Open the health monitoring log stream.
XM_hm_read	Read a health monitoring log entry.
XM_hm_seek	Sets the read position in the health monitoring stream.
XM_hm_status	Get the status of the health monitoring log stream.
XM_ia32_set_idt_desc	Sets an Interrupt Descriptor Table entry.
XM_ia32_update_sys_struct	Update a processor control register.
XM_idle_self	Idles the execution of the calling partition.
XM_iret	Return from a interrupt.
XM_lazy_hypercall	Execute a sequence of hypercalls.
XM_lazy_ia32_update_sys_struct	
XM_lazy_set_page_type	Set the type of a page.
XM_lazy_update_page32	
XM_lazy_write_register32	Modify a processor control register of 32-bit width.
XM_lazy_write_register64	Modify a processor control register of 64-bit width.
XM_mask_irq	Mask interrupt.
XM_memory_copy	Copy copies data from/to address spaces.
XM_multicall	Execute a sequence of hypercalls.
XM_override_trap_hndl	Override a trap handler entry.
XM_params_get_PCT	Return the address of the PCT.
XM_raise_ipvi	Raise an extended interrupt.
XM_read_object	Performs a read on a object.
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_reset_partition	Reset a partition.
XM_reset_system	Reset the system.
XM_resume_partition	Resume the execution of a partition.
XM_seek_object	Performs a seek on a object.
XM_send_queuing_message	Send a message in the specified queuing port.
XM_set_page_type	Changes the type of the physical page `pAddr` to `type`.
XM_set_partition_opmode	Set the Partition Operation mode (TBD)
XM_set_plan	Request a plan switch at the end of the current MAF.
XM_set_spare_guest	Set spare guest partition.
XM_set_timer	Arm a timer.
XM_shutdown_partition	Send a shutdown interrupt to a partition.
XM_suspend_partition	Suspend the execution of a partition.
XM_system_get_status	Get the current status of the system.
XM_trace_event	Records a trace entry.
XM_trace_open	Open a trace stream.
XM_trace_read	Read a trace event.
XM_trace_seek	Sets the read position in a trace stream.
XM_trace_status	Get the status of a trace stream.
XM_unmask_irq	Unmask interrupt.
XM_update_page32	Writes `val` in `pAddr`.
XM_write_console	Print a string in the hypervisor console.
XM_write_object	Performs a write on a object.
XM_write_register32	Modify a processor control register.
XM_write_sampling_message	Writes a message in the specified sampling port.

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 the partition developer. The remaining sections contain material for 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.
XtratuM follows the next 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 has to be stored in big-endian order, that is, the most significant byte standing at the lower address (0x..00) and the least significant byte standing to the upper address (0x..03).
The "C" declaration which meets these definitions is presented in the next listing:

```
// 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/ia32/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 following macros 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 6.2: /core/include/xmef.h

6.2 Hypercall mechanism

An 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:

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.
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.

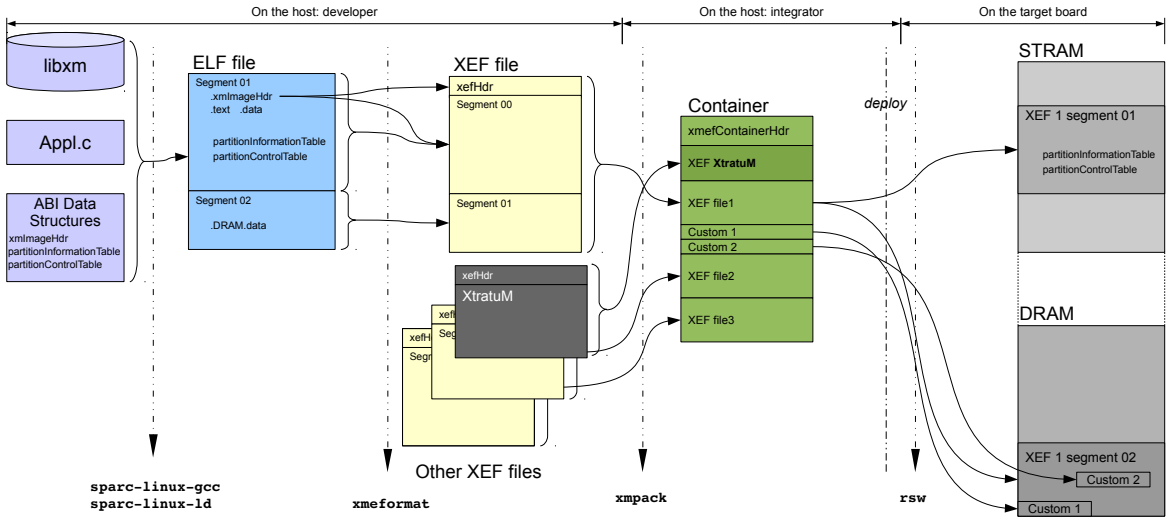


Figure 6.1: Executable formats.

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 {
    xm_u32_t signature;
    xm_u32_t xmAbiVersion; // XM's abi version
    xm_u32_t xmApiVersion; // XM's api version
    xm_u32_t imageId;    // To bind this image with the configuration
    xm_u32_t checksum; // header's checksum
    xmAddress_t sAddr; // partition's start memory address
    xmAddress_t eAddr; // partition's end memory address
    union {
        xmAddress_t ePoint; // XtratuM's entry point
        struct xmPartitionHdr *defaultPartitionHdr; // or partition Hdr.
    } entry;
    xm_u32_t noModules;
    struct xefCustomFile moduleTab[CONFIG_MAX_NO_FILES];
};
```

Listing 6.3: /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 2
#define XM_ABI_SUBVERSION 0
#define XM_ABI_REVISION 0

#define XM_API_VERSION 2
#define XM_API_SUBVERSION 0
#define XM_API_REVISION 0
```

Listing 6.4: /core/include/hypercalls.h

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

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

customFileTab: Table information about the customisation files.

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

Listing 6.5: /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 partition is only allowed to read.

```
typedef struct {
    xm_u32_t magic;
    xm_u32_t resetCounter;
    xm_u32_t resetStatus;

    xmAtomic_t iFlags;
    // BIT: 23..16: ARCH
    //      1: TRAP PENDING
```

```
    //      0: IRQ
    xmAtomic_t hwIrqsPend; // pending hw irqs
    xmAtomic_t hwIrqsMask; // masked hw irqs

    xmAtomic_t extIrqsPend; // pending extended irqs
    xmAtomic_t extIrqsMask; // masked extended irqs

    xmAtomic_t objDescClassPend; // Object descritors

    struct pctArch arch;
    struct {
        xm_u32_t noSlot:16, reserved:16;
        xm_u32_t id;
        xm_u32_t slotDuration;
        xm_u32_t slotUsed;
        xm_u32_t slotAccum;
    } schedInfo;
```

Listing 6.6: /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 {
    //xm_u32_t dbreg[8];
    pseudoDesc_t gdtr;
    pseudoDesc_t idtr;
    volatile xm_u32_t tr;
    volatile xm_u32_t cr4;
    volatile xm_u32_t cr3;
    volatile xm_u32_t cr2;
    volatile xm_u32_t cr0;
    struct {
        volatile xm_u32_t sAddr;
        volatile xm_u32_t eAddr;
    } atomicArea;
};
```

Listing 6.7: /core/include/ia32/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 an header (see listing 6.8) and a set of *segments*. The segments, like in ELF, represent blocks of memory that will be loaded in RAM.
The tool `xmeformat` converts from ELF or plain data files to XEF format, see chapter 8.

```
struct xefHdr {
#define XEFSIGNATURE 0x24584546
    xm_u32_t signature;
    xm_u32_t imageId;    // To bind this image with its configuration
    xmAddress_t secOffset;
    xm_s32_t nSec;
    xm_s32_t memSz;
    xmAddress_t relOffset;
    xm_s32_t nRel;
    xmAddress_t entry;
};
```

Listing 6.8: /core/include/xmef.h

signature: A 4 bytes word to identify the file as an 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. 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 6.9) which describes each one of the sections of the image (custom data XEF files have only one section).

```
struct xefSection {
    xmAddress_t pAddr;
    xmAddress_t vAddr;
    //xm_u32_t memSz;
    xm_u32_t fileSz;
    xmAddress_t offset;
};
```

Listing 6.9: /core/include/xmef.h

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

fileSize: The size of the segment within the file. This size could be different 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 8.

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. 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 several the file metadata (name of the files) and the content of each file. Also, which file contans the executable image and the customisation data of each partition is specified.

The container has 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 configuration files.

The container header has the following fields:

```
struct xmefPackageHeader {
    xm_u32_t signature;
    xm_u32_t version;
    xm_u32_t checksum;
    xm_u32_t noComponents; // Number of components.
    xmAddress_t componentOffset; // Offset to the table of components area.
    xm_u32_t noFiles;      // Number of files in the container.
    xmAddress_t fileTabOffset; // Offset to the table of files.
    xmSize_t fileDataLen;      // Length of the file data area.
    xmAddress_t fileDataOffset; // Offset to files data content.
    xmSize_t strTabLen;     // Length of the string area.
    xmAddress_t strTabOffset; // Offset to the string area.
};
```

Listing 6.10: /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 xmefComponent {
    xm_u32_t flags;
#define CONFIG_COMP_HYPERVISOR_FLAG 0x1
#define CONFIG_COMP_LOAD_FLAG 0x2
    xm_u32_t fileTabEntry;
    xm_s32_t noFiles;
```

```
};
```

Listing 6.11: /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 fileNameOffset;
    xmSize_t fileSize;
    xmAddress_t offset;
    xmSize_t size;
};
```

Listing 6.12: /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 8.3.1 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

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.

7.1 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.

There has been major changes from version 2.2 to 3.x in the source code configuration.

Parameter	Type	Default value
Processor		
Board	choice	[PC]
Physical memory layout		
Hypervisor arch-dependent parameters		
Number of GDTs entries per partition	int	32
Idle loop C-State (0-5)	int	1
System Timer	choice	[PIT] [HPET]
HPET Vendor ID	hex	0x8086
Max. identifier length (B)	int	16
Debug support	bool	y

Continues...

Parameter	Type	Default value
Hypervisor parameters		
Enable early XM output	bool	y
Enable spare time dynamic scheduling	bool	n
Partition parameters		
Kernel stack size (KB)	int	8
Devices		
Enable VGA support	bool	n
Enable UART support	bool	n
Enable Winbond GPIO support	bool	n
Objects		
Enable XM/partition status accounting	bool	n
Verbose HM events	bool	y
Enable partition id symbol on console write	bool	n

Enable VGA support: Only for ia32 target.

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.

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.

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.

7.2 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		
Load address	hex	0x1000000
Container Load Address	hex	0x1002000
Stack size (KB)	int	8

7.3 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.

7.3.1 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:

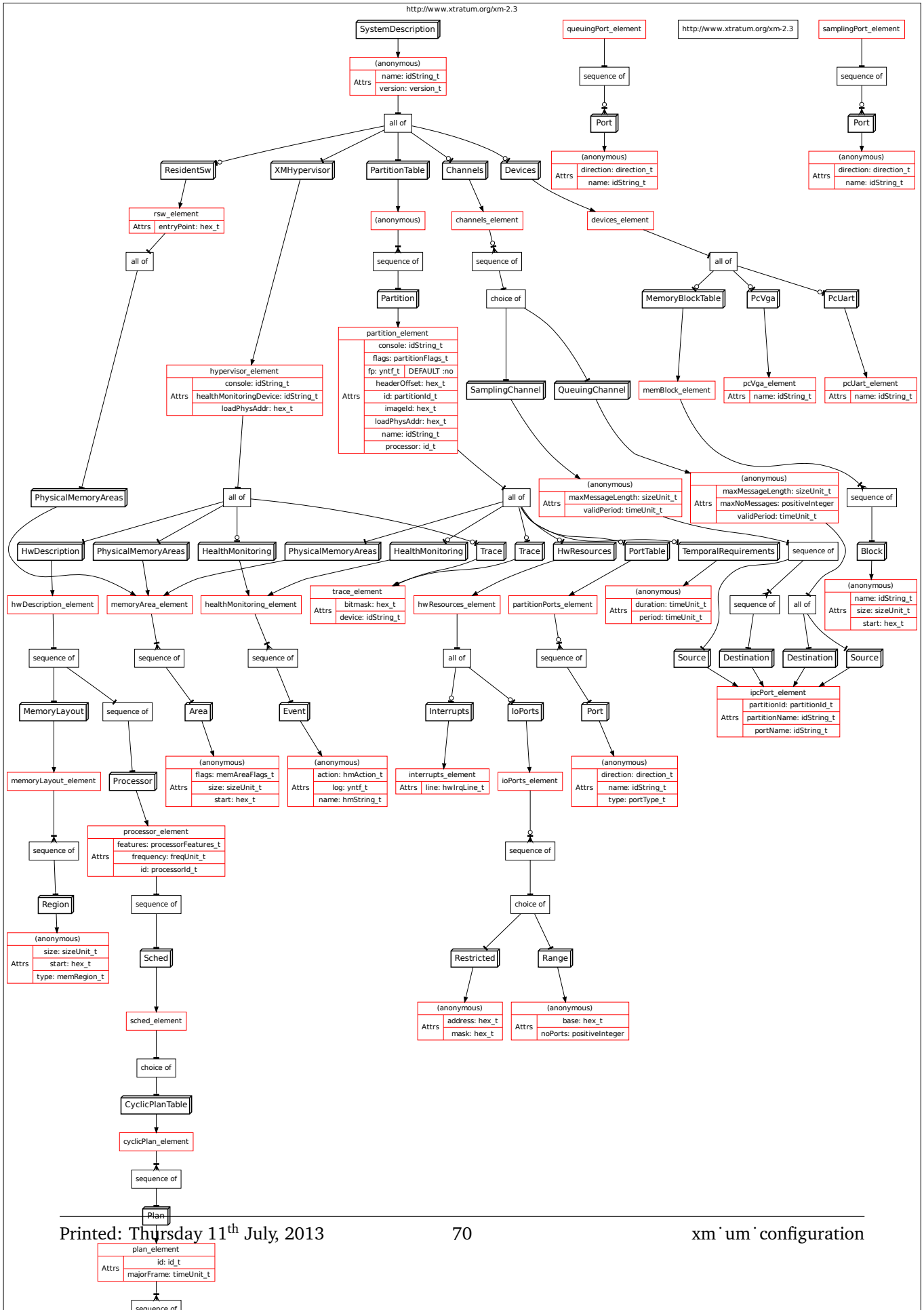
```
0xFfffFfff, 0x0, 0xF1, 0x80
```

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**.

./Trace/@bitmask Refers to the attribute **./@bitmask** of a **./Trace** element. The location of the element **./Trace** in the xml element hierarchy is relative to the context where the reference appears.


http://www.xtratum.org/xm-2.3
SystemDescription
(anonymous)
Attrs
name: idString_t
version: version_t
all of
ResidentSw
XMHypervisor
PartitionTable
Channels
Devices
rsw_element
Attrs
entryPoint: hex_t
all of
hypervisor_element
Attrs
console: idString_t
healthMonitoringDevice: idString_t
loadPhysAddr: hex_t
(anonymous)
sequence of
Partition
partition_element
Attrs
console: idString_t
flags: partitionFlags_t
fp: yntf_t DEFAULT :no
headerOffset: hex_t
id: partitionId_t
imageId: hex_t
loadPhysAddr: hex_t
name: idString_t
processor: id_t
channels_element
sequence of
choice of
SamplingChannel
QueuingChannel
queuingPort_element
sequence of
Port
(anonymous)
Attrs
direction: direction_t
name: idString_t
devices_element
all of
MemoryBlockTable
PcVga
PcUart
memBlock_element
pcVga_element
Attrs
name: idString_t
pcUart_element
Attrs
name: idString_t
http://www.xtratum.org/xm-2.3
samplingPort_element
sequence of
Port
(anonymous)
Attrs
direction: direction_t
name: idString_t
PhysicalMemoryAreas
all of
HwDescription
PhysicalMemoryAreas
HealthMonitoring
PhysicalMemoryAreas
HealthMonitoring
Trace
hwDescription_element
sequence of
MemoryLayout
sequence of
memoryLayout_element
Processor
processor_element
Attrs
features: processorFeatures_t
frequency: freqUnit_t
id: processorId_t
sequence of
Sched
sched_element
choice of
CyclicPlanTable
cyclicPlan_element
sequence of
Plan
plan_element
Attrs
id: id_t
majorFrame: timeUnit_t
sequence of
memoryArea_element
sequence of
Area
(anonymous)
Attrs
flags: memAreaFlags_t
size: sizeUnit_t
start: hex_t
healthMonitoring_element
sequence of
Event
(anonymous)
Attrs
action: hmAction_t
log: yntf_t
name: hmString_t
all of
Trace
HwResources
PortTable
TemporalRequirements
trace_element
Attrs
bitmask: hex_t
device: idString_t
hwResources_element
all of
Interrupts
IoPorts
interrupts_element
Attrs
line: hwIrqLine_t
ioPorts_element
sequence of
choice of
Restricted
Range
(anonymous)
Attrs
address: hex_t
mask: hex_t
(anonymous)
Attrs
base: hex_t
noPorts: positiveInteger
partitionPorts_element
sequence of
Port
(anonymous)
Attrs
direction: direction_t
name: idString_t
type: portType_t
(anonymous)
Attrs
duration: timeUnit_t
period: timeUnit_t
(anonymous)
Attrs
maxMessageLength: sizeUnit_t
validPeriod: timeUnit_t
(anonymous)
Attrs
maxMessageLength: sizeUnit_t
maxNoMessages: positiveInteger
validPeriod: timeUnit_t
sequence of
Block
(anonymous)
Attrs
name: idString_t
size: sizeUnit_t
start: hex_t
sequence of
Source
Destination
all of
Destination
Source
ipcPort_element
Attrs
partitionId: partitionId_t
partitionName: idString_t
portName: idString_t
(anonymous)
Attrs
size: sizeUnit_t
start: hex_t
type: memRegion_t
Region

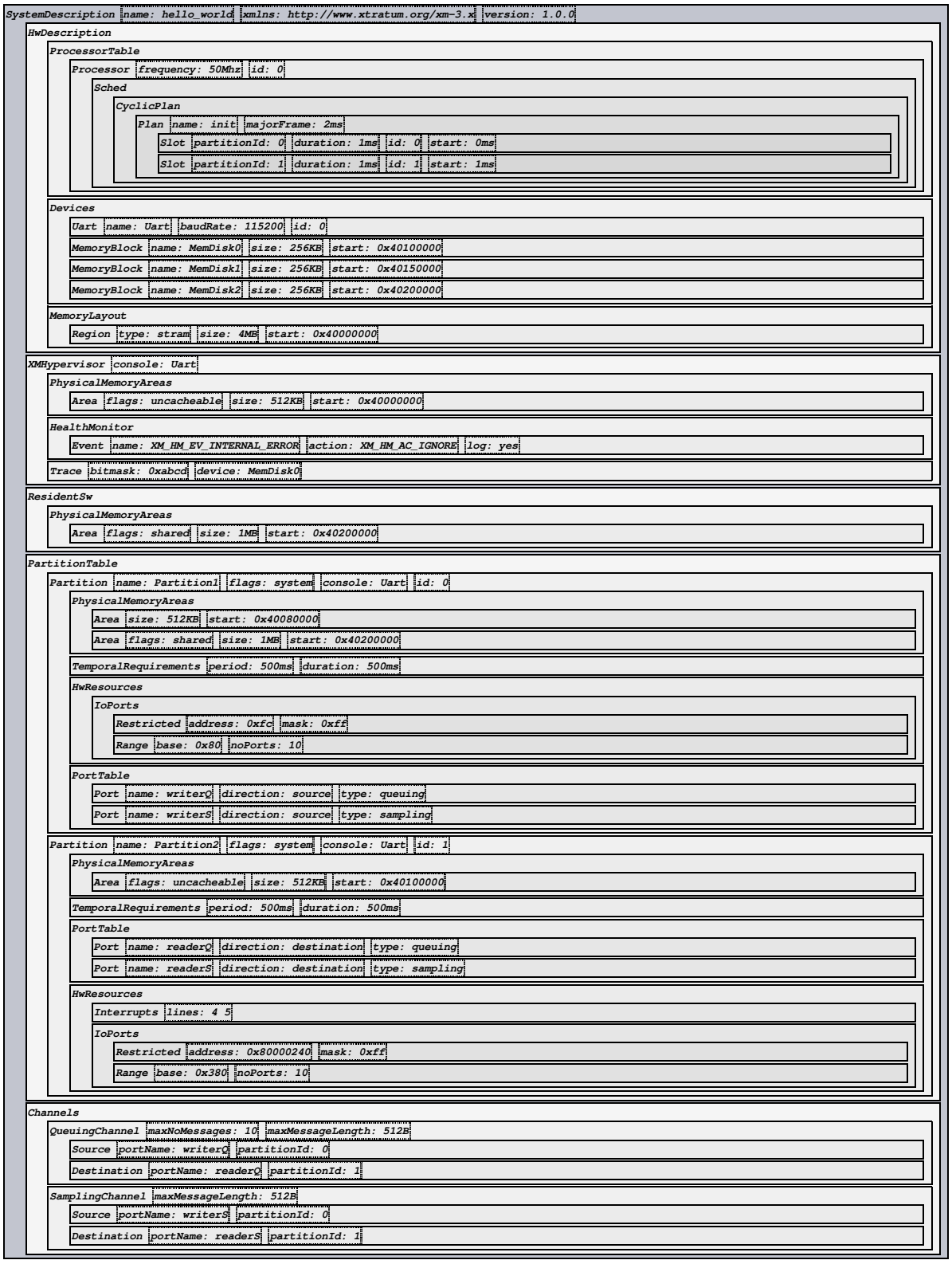


Figure 7.2: Graphical view of an example XM_CF configuration file.

7.3.2 The root element: /SystemDescription

Figure 7.1 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 7.2 on page 71 is a compact graphical representation of the nested structure a sample XM_CF configuration file. 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.

7.3.3 The /SystemDescription/XMHypervisor element

There are two optional attributes **./@console** and **./@healthMonitoringDevice**. The values of these attributes 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".

./Trace Defines where to store the traces messages emitted by XtratuM (the value of the attribute **./@device** shall be a the name of a device defined in **/SystemDescription/Devices**); and the hexadecimal bit mask to filter out which traces will not be stored (**./@bitmask**).

A health monitoring event element contains the following attributes:

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

```
      <xs:enumeration value="XM_HM_EV_OVERRUN"/>
      <xs:enumeration value="XM_HM_EV_SCHED_ERROR"/>
      <xs:enumeration value="XM_HM_EV_WATCHDOG_TIMER"/>
      <xs:enumeration value="XM_HM_EV_INCOMPATIBLE_INTERFACE"/>

#ifdef ia32
    <xs:enumeration value="XM_HM_EV_IA32_DIVIDE_EXCEPTION"/>
    <xs:enumeration value="XM_HM_EV_IA32_DEBUGGER_EXCEPTION"/>
    <xs:enumeration value="XM_HM_EV_IA32_NMI_EXCEPTION"/>
    <xs:enumeration value="XM_HM_EV_IA32_BREAKPOINT_EXCEPTION"/>
    <xs:enumeration value="XM_HM_EV_IA32_OVERFLOW_EXCEPTION"/>
    <xs:enumeration value="XM_HM_EV_IA32_BOUNDS_EXCEPTION"/>
    <xs:enumeration value="XM_HM_EV_IA32_INVALID_OPCODE"/>
    <xs:enumeration value="XM_HM_EV_IA32_COPROCESOR_UNAVAILABLE"/>
```

```
    <xs:enumeration value="XM_HM_EV_IA32_DOUBLE_FAULT"/>
    <xs:enumeration value="XM_HM_EV_IA32_COPROCESSOR_OVERRUN"/>
    <xs:enumeration value="XM_HM_EV_IA32_INVALID_TSS"/>
    <xs:enumeration value="XM_HM_EV_IA32_SEGMENT_NOT_PRESENT"/>
    <xs:enumeration value="XM_HM_EV_IA32_STACK_FAULT"/>
    <xs:enumeration value="XM_HM_EV_IA32_GENERAL_PROTECTION_FAULT"/>
    <xs:enumeration value="XM_HM_EV_IA32_PAGE_FAULT"/>
    <xs:enumeration value="XM_HM_EV_IA32_RESERVED"/>
    <xs:enumeration value="XM_HM_EV_IA32_MATH_FAULT"/>
    <xs:enumeration value="XM_HM_EV_IA32_ALIGNMENT_CHECK"/>
    <xs:enumeration value="XM_HM_EV_IA32_MACHINE_CHECK"/>
    <xs:enumeration value="XM_HM_EV_IA32_FLOATING_POINT_EXCEPTION"/>
#endif
      <!-- </track id="xml-list-hm-events" > -->
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="hmAction_t">
    <xs:restriction base="xs:string">
      <!-- <track id="xml-list-hm-actions" > -->
      <xs:enumeration value="XM_HM_AC_IGNORE"/>
      <xs:enumeration value="XM_HM_AC_SHUTDOWN"/>
      <xs:enumeration value="XM_HM_AC_COLD_RESET"/>
      <xs:enumeration value="XM_HM_AC_WARM_RESET"/>
      <xs:enumeration value="XM_HM_AC_SUSPEND"/>
      <xs:enumeration value="XM_HM_AC_HALT"/>
      <xs:enumeration value="XM_HM_AC_PROPAGATE"/>
      <!-- </track id="xml-list-hm-actions" > -->
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="portType_t">
    <xs:restriction base="xs:string">
      <xs:enumeration value="queuing"/>
      <xs:enumeration value="sampling"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:simpleType name="direction_t">
    <xs:restriction base="xs:string">
      <xs:enumeration value="source"/>
```

Listing 7.1: /user/tools/xmcparser/xmc.xsd.S

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

```
      <xs:enumeration value="no"/>
      <xs:enumeration value="true"/>
      <xs:enumeration value="false"/>
    </xs:restriction>
  </xs:simpleType>

  <!-- End Type -->
```

Listing 7.2: /user/tools/xmcparser/xmc.xsd.S

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

7.3.4 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). 1390

Also, the `./ProcessorTable/Processor` element defines the scheduling plan of this processor. It is specified in the element `./Processor/Sched/CyclicPlan/Plan/`[1]. The `./Plan` element has the required attributes `./@name` and `./majorFrame`; 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.

7.3.5 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.
Also the attribute `./@entryPoint` is used by XtratuM in the case of a cold system reset. In that case, XtratuM will give back the control of the system to the resident softwware by jumping to this address.

7.3.6 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:

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

fp If set, the partition is allowed to use floating point operations. By default not set.

sv If set, the partition has system privileges. By default not set.

./@boot Boolean attribute. 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.

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.
shared	It is allowed to map this area in other partitions.
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.

7.3.7 The `/SystemDescription/Channels` element

This is an optional element with no attributes and which contains a list of channel elements. There are two types of channels:

`./SamplingChannel` Shall contain one **`./Source`** element and one or more **`./Destination`** 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.

`./QueuingChannel` Shall contain one **`./Source`** element and one **`./Destination`** 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.

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 XML schema which defines the configuration file is in the appendix A.

Chapter 8

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.

8.1 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.

8.1.1 xmcparser

Compiles XtratuM XML configuration files

SYNOPSIS

xmcparser [-s xsd_file] [-o output_file] XM_CF.xml

DESCRIPTION

`xmcparser` reads an XtratuM XML configuration file and transforms it into a set of ”C” data structures initialized with the XML data. `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.

OPTIONS

-d

Prints the dafault XML schema used to validate the XML configuration file.

-o file

Place output in file.

-s xsd_file

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

USAGE EXAMPLES

```
xmcparser -o xm_cf.c.xmc xm_cf.xml
```

8.2 ELF to XEF (`elf2xef`)

8.2.1 elf2xef

Converts ELF files to XEF format

SYNOPSIS

elf2xef [-o outfile] [-i <id>] [-x <section_id>] infile

DESCRIPTION

`elf2xef` converts an ELF 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).
An XEF file has a header and a set of segments. The segments corresponds to the allocatable sections of the source ELF file.

-i <id>

Set the identification number of the outfile to <id>.

-o outfile

Places output in file outfile.

-x <section_id>

Exclude the ELF section with id <section_id> from the output file outfile.

INPUT FILE FORMAT

The input file should conform to the ELF32 version 1 format, refer to the ELF specification document for more details.

```
ELF Header:
  Class:                             ELF32
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Flags:                             0x0
```

USAGE EXAMPLES

Create a partition file:

```
$ elf2xef -o partition.xef partition.elf
```

Build the hypervisor XEF file:

```
$ elf2xef -o xm_core.xef -c core/xm_core
```

8.3 Container builder (xmpack)

8.3.1 xmpack

Manage the XtratuM system image container

SYNOPSIS

xmpack build [{-h|-b|-p} *file*[:*file*...] ...] *container*
xmpack list -c *container*
xmpack extract [-*o file*] -c *noComponent* -f *noFile 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 partitions.
The container file should be written in ROM. 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 files. A component is used to store an executable unit, which can be: the XtratuM hypervisor, a bootable partition or a non-bootable 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). The rest of files of the components are optional

and can be used to attach extra data (for example the configuration table for the XtratuM component or the customization file for partition ones).

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.

[build]

A new container is created. Three kind of components can be included:

-h to create an [H]ypervisor component;

-b to create a [B]ootable partition; and

-p to create a non-bootable [P]artition.

The files that are part of each component are specified as a list separated by "**:**".

[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.

[extract]

Writes in the output file the specified file. The file is identified as the component number (parameter **-c**) and the file within the component (parameter **-f**).

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 -h ../core/xm_core.bin:xm_ct.bin -b partition1.bin -o container
```

The same example but the second container has now two files: the partition image and a customisation file:

```
$ xmpack build -h ../core/xm_core.bin:xm_cf.bin \
               -b partition1.bin:p1.cfg \
                           -b 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>
```

8.4 Bootable image creator (`rswbuild`)

8.4.1 rswbuild

Create a bootable image

SYNOPSIS

rswbuild container 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
```

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-2.3"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns="http://www.xtratum.org/xm-2.3"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">
  <xs:include schemaLocation="ia32/xmc_ia32.xsd"/>
  <xs:include schemaLocation="devices/xmc_pc_vga.xsd"/>
  <xs:include schemaLocation="devices/xmc_pc_uart.xsd"/>
 <xs:include schemaLocation="devices/xmc_memblock.xsd"/>
  <!-- Basic types definition -->
  <xs:simpleType name="id_t">
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="0"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="partitionId_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:maxLength value="16"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="processorId_t">
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="0"/>
      <xs:maxExclusive value="256"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="hwIrqId_t">
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="0"/>
```

```
37        <xs:maxExclusive value="16"/>
38      </xs:restriction>
39    </xs:simpleType>
40    <xs:simpleType name="hwIrqLine_t">
41      <xs:list itemType="hwIrqId_t"/>
42    </xs:simpleType>
43    <xs:simpleType name="hex_t">
44      <xs:restriction base="xs:string">
45        <xs:pattern value="0x[0-9a-fA-F]+"/>
46      </xs:restriction>
47    </xs:simpleType>
48    <xs:simpleType name="version_t">
49      <xs:restriction base="xs:string">
50        <xs:pattern value="[0-9]+.[0-9]+.[0-9]+"/>
51      </xs:restriction>
52    </xs:simpleType>
53    <xs:simpleType name="freqUnit_t">
54      <xs:restriction base="xs:string">
55        <xs:pattern value="[0-9]+(.[0-9]+)?([MK][Hh]z)"/>
56      </xs:restriction>
57    </xs:simpleType>
58    <xs:simpleType name="processorFeatures_t">
59      <xs:list itemType="cpuFeatList_t"/>
60    </xs:simpleType>
61    <xs:simpleType name="partitionFlagsEnum_t">
62      <xs:restriction base="xs:string">
63        <xs:enumeration value="sv"/>
64        <xs:enumeration value="boot"/>
65      </xs:restriction>
66    </xs:simpleType>
67    <xs:simpleType name="partitionFlags_t">
68      <xs:list itemType="partitionFlagsEnum_t"/>
69    </xs:simpleType>
70    <xs:simpleType name="memAreaFlagsEnum_t">
71      <xs:restriction base="xs:string">
72        <xs:enumeration value="shared"/>
73        <xs:enumeration value="mapped"/>
74        <xs:enumeration value="read"/>
75        <xs:enumeration value="write"/>
76        <xs:enumeration value="exec"/>
77        <xs:enumeration value="rom"/>
78        <xs:enumeration value="flag0"/>
79        <xs:enumeration value="flag1"/>
80        <xs:enumeration value="flag2"/>
81        <xs:enumeration value="flag3"/>
82      </xs:restriction>
83    </xs:simpleType>
84    <xs:simpleType name="memAreaFlags_t">
85      <xs:list itemType="memAreaFlagsEnum_t"/>
86    </xs:simpleType>
87    <xs:simpleType name="sizeUnit_t">
88      <xs:restriction base="xs:string">
89        <xs:pattern value="[0-9]+(.[0-9]+)?([MK]?B)"/>
90      </xs:restriction>
```

```
91    </xs:simpleType>
92    <xs:simpleType name="timeUnit_t">
93      <xs:restriction base="xs:string">
94        <xs:pattern value="[0-9]+(.[0-9]+)?([mu]?[sS])"/>
95      </xs:restriction>
96    </xs:simpleType>
97    <xs:simpleType name="hmString_t">
98      <xs:restriction base="xs:string">
99        <!-- <track id="xml-list-hm-events" > -->
100       <xs:enumeration value="XM_HM_EV_INTERNAL_ERROR"/>
101       <xs:enumeration value="XM_HM_EV_UNEXPECTED_TRAP"/>
102       <xs:enumeration value="XM_HM_EV_PARTITION_ERROR"/>
103       <xs:enumeration value="XM_HM_EV_PARTITION_INTEGRITY"/>
104       <xs:enumeration value="XM_HM_EV_MEM_PROTECTION"/>
105       <xs:enumeration value="XM_HM_EV_OVERRUN"/>
106       <xs:enumeration value="XM_HM_EV_SCHED_ERROR"/>
107       <xs:enumeration value="XM_HM_EV_WATCHDOG_TIMER"/>
108       <xs:enumeration value="XM_HM_EV_INCOMPATIBLE_INTERFACE"/>
109     <xs:enumeration value="XM_HM_EV_IA32_DIVIDE_EXCEPTION"/>
110     <xs:enumeration value="XM_HM_EV_IA32_DEBUGGER_EXCEPTION"/>
111     <xs:enumeration value="XM_HM_EV_IA32_NMI_EXCEPTION"/>
112     <xs:enumeration value="XM_HM_EV_IA32_BREAKPOINT_EXCEPTION"/>
113     <xs:enumeration value="XM_HM_EV_IA32_OVERFLOW_EXCEPTION"/>
114     <xs:enumeration value="XM_HM_EV_IA32_BOUNDS_EXCEPTION"/>
115     <xs:enumeration value="XM_HM_EV_IA32_INVALID_OPCODE"/>
116     <xs:enumeration value="XM_HM_EV_IA32_COPROCESOR_UNAVAILABLE"/>
117     <xs:enumeration value="XM_HM_EV_IA32_DOUBLE_FAULT"/>
118     <xs:enumeration value="XM_HM_EV_IA32_COPROCESSOR_OVERRUN"/>
119     <xs:enumeration value="XM_HM_EV_IA32_INVALID_TSS"/>
120     <xs:enumeration value="XM_HM_EV_IA32_SEGMENT_NOT_PRESENT"/>
121     <xs:enumeration value="XM_HM_EV_IA32_STACK_FAULT"/>
122     <xs:enumeration value="XM_HM_EV_IA32_GENERAL_PROTECTION_FAULT"/>
123     <xs:enumeration value="XM_HM_EV_IA32_PAGE_FAULT"/>
124     <xs:enumeration value="XM_HM_EV_IA32_RESERVED"/>
125     <xs:enumeration value="XM_HM_EV_IA32_MATH_FAULT"/>
126     <xs:enumeration value="XM_HM_EV_IA32_ALIGNMENT_CHECK"/>
127     <xs:enumeration value="XM_HM_EV_IA32_MACHINE_CHECK"/>
128     <xs:enumeration value="XM_HM_EV_IA32_FLOATING_POINT_EXCEPTION"/>
129       <!-- </track id="xml-list-hm-events" > -->
130     </xs:restriction>
131   </xs:simpleType>
132   <xs:simpleType name="hmAction_t">
133     <xs:restriction base="xs:string">
134       <!-- <track id="xml-list-hm-actions" > -->
135       <xs:enumeration value="XM_HM_AC_IGNORE"/>
136       <xs:enumeration value="XM_HM_AC_SHUTDOWN"/>
137       <xs:enumeration value="XM_HM_AC_COLD_RESET"/>
138       <xs:enumeration value="XM_HM_AC_WARM_RESET"/>
139       <xs:enumeration value="XM_HM_AC_SUSPEND"/>
140       <xs:enumeration value="XM_HM_AC_HALT"/>
141       <xs:enumeration value="XM_HM_AC_PROPAGATE"/>
142       <!-- </track id="xml-list-hm-actions" > -->
143     </xs:restriction>
144   </xs:simpleType>
```

```
145   <xs:simpleType name="portType_t">
146     <xs:restriction base="xs:string">
147       <xs:enumeration value="queuing"/>
148       <xs:enumeration value="sampling"/>
149     </xs:restriction>
150   </xs:simpleType>
151   <xs:simpleType name="direction_t">
152     <xs:restriction base="xs:string">
153       <xs:enumeration value="source"/>
154       <xs:enumeration value="destination"/>
155     </xs:restriction>
156   </xs:simpleType>
157   <xs:simpleType name="yntf_t">
158     <xs:restriction base="xs:string">
159       <xs:enumeration value="yes"/>
160       <xs:enumeration value="no"/>
161       <xs:enumeration value="true"/>
162       <xs:enumeration value="false"/>
163     </xs:restriction>
164   </xs:simpleType>
165   <!-- End Type -->
166   <!-- Elements -->
167   <!-- Hypervisor -->
168   <xs:complexType name="hypervisor_element" mixed="false">
169     <xs:all>
170       <xs:element name="PhysicalMemoryAreas" type="memoryArea_element"/>
171       <xs:element name="HwDescription" type="hwDescription_element" />
172       <xs:element name="HealthMonitoring" type="healthMonitoring_element"
              minOccurs="0" />
173       <xs:element name="Trace" type="trace_element" minOccurs="0" />
174     </xs:all>
175     <xs:attribute name="console" type="idString_t" use="optional"/>
176     <xs:attribute name="healthMonitoringDevice" type="idString_t" use="optional"
          />
177     <xs:attribute name="loadPhysAddr" type="hex_t" use="required"/>
178   </xs:complexType>
179   <!-- Rsw -->
180   <xs:complexType name="rsw_element">
181     <xs:all>
182       <xs:element name="PhysicalMemoryAreas" type="memoryArea_element"/>
183     </xs:all>
184     <xs:attribute name="entryPoint" type="hex_t" use="optional"/>
185   </xs:complexType>
186   <!-- Partition -->
187   <xs:complexType name="partition_element" mixed="false">
188     <xs:all>
189       <xs:element name="PhysicalMemoryAreas" type="memoryArea_element"/>
190       <xs:element name="TemporalRequirements" minOccurs="0">
191  <xs:complexType>
192    <xs:attribute name="period" type="timeUnit_t" use="required"/>
193    <xs:attribute name="duration" type="timeUnit_t" use="required"/>
194  </xs:complexType>
195       </xs:element>
```

```
196       <xs:element name="HealthMonitoring" type="healthMonitoring_element"
              minOccurs="0" />
197       <xs:element name="HwResources" type="hwResources_element" minOccurs="0" />
198       <xs:element name="PortTable" type="partitionPorts_element" minOccurs="0" /
              >
199       <xs:element name="Trace" type="trace_element" minOccurs="0" />
200     </xs:all>
201     <xs:attribute name="id" type="partitionId_t" use="required"/>
202     <xs:attribute name="name" type="idString_t" use="optional"/>
203     <xs:attribute name="processor" type="id_t" use="required"/>
204     <xs:attribute name="flags" type="partitionFlags_t" use="optional"/>
205     <xs:attribute name="loadPhysAddr" type="hex_t" use="required"/>
206     <xs:attribute name="headerOffset" type="hex_t" use="required"/>
207     <xs:attribute name="fp" type="yntf_t" use="optional" default="no"/>
208     <xs:attribute name="imageId" type="hex_t" use="required"/>
209     <xs:attribute name="console" type="idString_t" use="optional"/>
210   </xs:complexType>
211   <!-- Trace -->
212   <xs:complexType name="trace_element">
213     <xs:attribute name="device" type="idString_t" use="required"/>
214     <xs:attribute name="bitmask" type="hex_t" use="required"/>
215   </xs:complexType>
216   <!-- Communication Ports -->
217   <xs:complexType name="partitionPorts_element">
218     <xs:sequence minOccurs="0" maxOccurs="unbounded">
219       <xs:element name="Port">
220 <xs:complexType>
221    <xs:attribute name="name" type="idString_t" use="required"/>
222    <xs:attribute name="direction" type="direction_t" use="required"/>
223    <xs:attribute name="type" type="portType_t" use="required"/>
224 </xs:complexType>
225       </xs:element>
226     </xs:sequence>
227   </xs:complexType>
228   <!-- Channels -->
229   <xs:complexType name="channels_element">
230     <xs:sequence minOccurs="0" maxOccurs="unbounded">
231       <xs:choice>
232       <xs:element name="SamplingChannel">
233 <xs:complexType>
234     <xs:sequence minOccurs="1">
235       <xs:element name="Source" type="ipcPort_element" />
236       <xs:sequence minOccurs="1" maxOccurs="unbounded">
237               <xs:element name="Destination" type="ipcPort_element"/>
238             </xs:sequence>
239     </xs:sequence>
240     <xs:attribute name="maxMessageLength" type="sizeUnit_t" use="required"/>
241     <xs:attribute name="validPeriod" type="timeUnit_t" use="optional"/>
242 </xs:complexType>
243       </xs:element>
244       <xs:element name="QueuingChannel">
245 <xs:complexType>
246    <xs:all minOccurs="1">
247      <xs:element name="Source" type="ipcPort_element" />
```

```
248             <xs:element name="Destination" type="ipcPort_element"/>
249     </xs:all>
250     <xs:attribute name="maxMessageLength" type="sizeUnit_t" use="required"/>
251     <xs:attribute name="maxNoMessages" type="xs:positiveInteger" use="required"/
            >
252     <xs:attribute name="validPeriod" type="timeUnit_t" use="optional"/>
253   </xs:complexType>
254       </xs:element>
255       </xs:choice>
256     </xs:sequence>
257   </xs:complexType>
258   <!-- Devices -->
259   <xs:complexType name="devices_element">
260     <xs:all>
261       <xs:element name="MemoryBlockTable" type="memBlock_element" minOccurs="0"
              />
262       <xs:element name="PcVga" type="pcVga_element" minOccurs="0" />
263       <xs:element name="PcUart" type="pcUart_element" minOccurs="0" />
264     </xs:all>
265   </xs:complexType>
266   <!-- IPC Port -->
267   <xs:complexType name="ipcPort_element">
268     <xs:attribute name="partitionId" type="partitionId_t" use="required"/>
269     <xs:attribute name="partitionName" type="idString_t" use="optional"/>
270     <xs:attribute name="portName" type="idString_t" use="required"/>
271   </xs:complexType>
272   <!-- Sampling port -->
273   <xs:complexType name="samplingPort_element">
274     <xs:sequence>
275       <xs:element name="Port" minOccurs="0" maxOccurs="unbounded">
276   <xs:complexType>
277     <xs:attribute name="name" type="idString_t" use="required"/>
278     <xs:attribute name="direction" type="direction_t" use="required"/>
279     <!--<xs:attribute name="maxMessageLength" type="sizeUnit_t"
280     use="required"/>-->
281   </xs:complexType>
282       </xs:element>
283     </xs:sequence>
284   </xs:complexType>
285   <!-- Queueing port -->
286   <xs:complexType name="queuingPort_element">
287       <xs:sequence>
288   <xs:element name="Port" minOccurs="0" maxOccurs="unbounded">
289     <xs:complexType>
290       <xs:attribute name="name" type="idString_t" use="required"/>
291       <xs:attribute name="direction" type="direction_t" use="required"/>
292       <!-- <xs:attribute name="maxMessageLength"
293       type="sizeUnit_t" use="required"/>
294     <xs:attribute name="maxNoMessages"
295       type="xs:positiveInteger" use="required"/>-->
296     </xs:complexType>
297   </xs:element>
298       </xs:sequence>
299   </xs:complexType>
```

```
300  <!-- Hw Description -->
301   <xs:complexType name="hwDescription_element">
302     <xs:sequence>
303     <xs:sequence minOccurs="1" maxOccurs="1">
304       <xs:element name="Processor" type="processor_element" />
305     </xs:sequence>
306     <xs:element name="MemoryLayout" type="memoryLayout_element"/>
307     </xs:sequence>
308   </xs:complexType>
309   <!-- Processor -->
310   <xs:complexType name="processor_element">
311     <xs:sequence minOccurs="1" maxOccurs="1">
312       <xs:element name="Sched" type="sched_element"/>
313     </xs:sequence>
314     <xs:attribute name="id" type="processorId_t" use="required"/>
315     <xs:attribute name="frequency" type="freqUnit_t" use="optional"/>
316     <xs:attribute name="features" type="processorFeatures_t" use="optional" />
317   </xs:complexType>
318   <!-- HwResource -->
319   <xs:complexType name="hwResources_element">
320     <xs:all>
321       <xs:element name="IoPorts" type="ioPorts_element" minOccurs="0" />
322       <xs:element name="Interrupts" type="interrupts_element" minOccurs="0" />
323     </xs:all>
324   </xs:complexType>
325   <!-- Io Ports -->
326   <xs:complexType name="ioPorts_element">
327     <xs:sequence minOccurs="0" maxOccurs="unbounded">
328       <xs:choice>
329  <xs:element name="Range">
330          <xs:complexType>
331      <xs:attribute name="base" type="hex_t" use="required"/>
332      <xs:attribute name="noPorts" type="xs:positiveInteger" use="required"/>
333          </xs:complexType>
334  </xs:element>
335  <xs:element name="Restricted">
336    <xs:complexType>
337      <xs:attribute name="address" type="hex_t" use="required"/>
338      <xs:attribute name="mask" type="hex_t" use="optional"/>
339    </xs:complexType>
340  </xs:element>
341       </xs:choice>
342     </xs:sequence>
343   </xs:complexType>
344   <!-- Hw Interrupts -->
345   <xs:complexType name="interrupts_element">
346   <xs:attribute name="line" type="hwIrqLine_t" use="required"/>
347   </xs:complexType>
348   <!-- Sched -->
349   <xs:complexType name="sched_element">
350     <xs:choice>
351       <xs:element name="CyclicPlanTable" type="cyclicPlan_element"/>
352       <!-- <xs:element name="FixPriority">
353  <xs:complexType>
```

```
354     <xs:attribute name="priority" type="id_t" use="required"/>
355   </xs:complexType>
356       </xs:element> -->
357     </xs:choice>
358    </xs:complexType>
359    <!-- CyclicPlan -->
360    <xs:complexType name="cyclicPlan_element">
361      <xs:sequence minOccurs="1" maxOccurs="unbounded">
362        <xs:element name="Plan" type="plan_element" />
363      </xs:sequence>
364    </xs:complexType>
365    <!-- Plan -->
366    <xs:complexType name="plan_element">
367      <xs:sequence minOccurs="1" maxOccurs="unbounded">
368        <xs:element name="Slot">
369   <xs:complexType>
370      <xs:attribute name="id" type="id_t" use="required"/>
371           <xs:attribute name="start" type="timeUnit_t" use="required"/>
372           <xs:attribute name="duration" type="timeUnit_t" use="required"/>
373           <xs:attribute name="partitionId" type="partitionId_t" use="required"/>
374         </xs:complexType>
375       </xs:element>
376     </xs:sequence>
377     <xs:attribute name="id" type="id_t" use="required"/>
378     <xs:attribute name="majorFrame" type="timeUnit_t" use="required"/>
379    </xs:complexType>
380    <!-- Health Monitoring -->
381    <xs:complexType name="healthMonitoring_element">
382      <xs:sequence minOccurs="1" maxOccurs="unbounded">
383        <xs:element name="Event">
384          <xs:complexType>
385     <xs:attribute name="name" type="hmString_t" use="optional"/>
386     <xs:attribute name="action" type="hmAction_t" use="required"/>
387     <xs:attribute name="log" type="yntf_t" use="required"/>
388   </xs:complexType>
389       </xs:element>
390     </xs:sequence>
391    </xs:complexType>
392    <!-- Memory Layout -->
393    <xs:complexType name="memoryLayout_element">
394      <xs:sequence minOccurs="1" maxOccurs="unbounded">
395        <xs:element name="Region">
396          <xs:complexType>
397     <xs:attribute name="type" type="memRegion_t" use="required"/>
398           <xs:attribute name="start" type="hex_t" use="required"/>
399           <xs:attribute name="size" type="sizeUnit_t" use="required"/>
400         </xs:complexType>
401       </xs:element>
402     </xs:sequence>
403    </xs:complexType>
404   <!-- Memory Area -->
405    <xs:complexType name="memoryArea_element">
406     <xs:sequence minOccurs="1" maxOccurs="unbounded">
407       <xs:element name="Area">
```

```
408         <xs:complexType>
409           <xs:attribute name="start" type="hex_t" use="required"/>
410           <xs:attribute name="size" type="sizeUnit_t" use="required"/>
411           <xs:attribute name="flags" type="memAreaFlags_t" use="optional" />
412         </xs:complexType>
413       </xs:element>
414     </xs:sequence>
415   </xs:complexType>
416   <!-- Root Element -->
417   <xs:element name="SystemDescription">
418     <xs:complexType>
419       <xs:all>
420   <xs:element name="XMHypervisor" type="hypervisor_element"/>
421   <xs:element name="ResidentSw" type="rsw_element" minOccurs="0" />
422   <xs:element name="PartitionTable">
423   <xs:complexType>
424           <xs:sequence maxOccurs="unbounded">
425      <xs:element name="Partition" type="partition_element" />
426    </xs:sequence>
427   </xs:complexType>
428   </xs:element>
429       <xs:element name="Channels" type="channels_element" minOccurs="0" />
430       <xs:element name="Devices" type="devices_element" minOccurs="0" />
431       </xs:all>
432       <xs:attribute name="version" type="version_t" use="required"/>
433       <xs:attribute name="name" type="idString_t" use="required"/>
434       </xs:complexType>
435     </xs:element>
436     <!-- End Root Element -->
437     <!-- Elements -->
438 </xs:schema>
```

Listing A.1: `xmc.xsd`

This page is intentionally left blank.

GNU Free Documentation License

Version 1.2, November 2002
Copyright © 2000,2001,2002 Free Software Foundation, Inc.

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

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. 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

customisation file A user defied 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 correct service.

fault A fault is the adjudged or hypothesized 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 Or peripheral port, 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 intructions.

native hardware The existing hardware: processor, interrupt, 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”. 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 sofware that is executed directly in ROM memory right after a system reboot, also refered as boot-loader or firmware. Among other tasks, it is in charge of loading in RAM memory XtratuM and the initial partitions.

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 control the system, and other partitions. Originally these partitions were named “supervisor partitions” but to avoid confusion with the processor modes it was 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.
DMA	Direct Memory Access.
ELF	Executable and Linkable Format.
FIFO	First In First Out.
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.
ST	Security Target.
TBR	Trap Base Register. A special LEON2 register.
TSC	TSF Scope of Control.
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.

Index