

# Software Reference Manual

## XM-ARM

---

Fent Innovative Software Solutions

September, 2016

Reference: 14-035-03.006.sum.03



This page is intentionally left blank.

# DOCUMENT CONTROL PAGE

**TITLE:** Software Reference Manual: XM-ARM

**AUTHOR/S:** Fent Innovative Software Solutions

**LAST PAGE NUMBER:** 81

**VERSION OF SOURCE CODE:** XM-ARMv2.0.4 for processor ARM CORTEX-A9()

**REFERENCE ID:** 14-035-03.006.sum.03

**SUMMARY:** This document contains the set of manual pages of XtratuM.

**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 {14-035-03.006.sum.03,
  title = {Software Reference Manual: XM-ARM},
  author = { Fent Innovative Software Solutions},
  institution = {Fent Innovative Software Solutions, S.L.},
  number = {14-035-03.006.sum.03},
  year={September, 2016},
}
```

**Copyright © September, 2016** Fent Innovative Software Solutions, S.L.

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

## Changes:

Version	Date	Author	Comments
01	09/06/2015	Javier Coronel	Initial Software Reference Manual for XM-ARMv2.0.0
02	12/01/2016	Manuel Muñoz and Javier Coronel	Some hypercalls have been updated: XM_clear_irqmask.xml XM_clear_irqpend XM_set_irqmask XM_set_irqpend XM_set_timer XM_create_queueing_port XM_create_sampling_port XM_get_gid_by_name XM_get_partition_mmap XM_get_partition_status

Version	Date	Author	Comments
			XM.get_plan_status XM.get_queuing_port_info XM.get_queuing_port_status XM.get_sampling_port_info XM.get_sampling_port_status XM.get_system_status XM.get_time XM.get_vcpuid XM.halt_partition XM.halt_system XM.halt_vcpu XM.hm.raise_event XM.hm.read XM.hm.status XM.idle_self XM.memory_copy XM.params.get_PCT XM.read_sampling_message XM.receive_queuing_message XM.reset_partition XM.reset_system XM.reset_vcpu XM.resume_imm_partition XM.resume_partition XM.send_queuing_message XM.set_cache_state XM.shutdown_partition XM.suspend_partition XM.switch_imm_sched_plan XM.switch_sched_plan XM.trace_event XM.trace_read XM.trace_status XM.write_console XM.write_sampling_message init_libxm XEF_parse_file.xml Changes for XM-ARMv2.0.3
03	09/09/2016	Manuel Muñoz and Javier Coronel	Some hypercalls have been updated: XM.clear_irqmask XM.clear_irqpend XM.reset_partition XM.reset_system Changes for XM-ARMv2.0.4

# Contents

<b>Preface</b>	<b>ix</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Global elements . . . . .	1
1.1.1 Basic types . . . . .	1
1.1.2 Extended types . . . . .	1
1.1.3 Return codes . . . . .	2
1.1.4 Variables . . . . .	2
1.1.5 API and ABI version numbers . . . . .	2
1.2 System management types . . . . .	2
1.2.1 System types . . . . .	2
1.3 Partition management types . . . . .	3
1.3.1 Partition types . . . . .	3
1.3.2 Partition Control Table (PCT) . . . . .	4
1.3.3 Memory Map Table (MMT) . . . . .	5
1.4 Time management variables . . . . .	6
1.4.1 Time management variables . . . . .	6
1.5 Memory management type . . . . .	6
1.5.1 Memory management types . . . . .	6
1.6 Plan management type . . . . .	6
1.6.1 Plan management type . . . . .	6
1.7 Inter-partition communication types . . . . .	7
1.7.1 Sampling ports . . . . .	7
1.7.2 Queuing ports . . . . .	7
1.8 Health Monitor types . . . . .	8
1.8.1 Health Monitor types . . . . .	8
1.8.2 HM actions . . . . .	8
1.8.3 HM events . . . . .	9
1.9 Trace management types . . . . .	9

1.9.1	Trace types . . . . .	9
1.10	Interrupt management types . . . . .	10
1.10.1	Interrupt management symbols . . . . .	10
<b>2</b>	<b>Hypercalls</b>	<b>11</b>
2.1	System Management . . . . .	12
2.1.1	XM_get_system_status . . . . .	12
2.1.2	XM_halt_system . . . . .	13
2.1.3	XM_reset_system . . . . .	14
2.2	Partition Management . . . . .	15
2.2.1	XM_get_partition_mmap . . . . .	15
2.2.2	XM_get_partition_status . . . . .	16
2.2.3	XM_halt_partition . . . . .	17
2.2.4	XM_idle_self . . . . .	18
2.2.5	XM_params_get_PCT . . . . .	20
2.2.6	XM_reset_partition . . . . .	21
2.2.7	XM_resume_imm_partition . . . . .	23
2.2.8	XM_resume_partition . . . . .	24
2.2.9	XM_shutdown_partition . . . . .	25
2.2.10	XM_suspend_partition . . . . .	26
2.3	Multicore Management . . . . .	27
2.3.1	XM_get_vcpuid . . . . .	27
2.3.2	XM_halt_vcpu . . . . .	28
2.3.3	XM_reset_vcpu . . . . .	29
2.4	Time Management . . . . .	30
2.4.1	XM_get_time . . . . .	30
2.4.2	XM_set_timer . . . . .	32
2.5	Plan Management . . . . .	34
2.5.1	XM_get_plan_status . . . . .	34
2.5.2	XM_switch_imm_sched_plan . . . . .	35
2.5.3	XM_switch_sched_plan . . . . .	36
2.6	Inter-Partition Communication . . . . .	37
2.6.1	XM_create_queueing_port . . . . .	37
2.6.2	XM_create_sampling_port . . . . .	39
2.6.3	XM_get_commpport_bitmap . . . . .	40
2.6.4	XM_get_queueing_port_info . . . . .	41
2.6.5	XM_get_queueing_port_status . . . . .	42

2.6.6	XM_get_sampling_port_info . . . . .	43
2.6.7	XM_get_sampling_port_status . . . . .	44
2.6.8	XM_read_sampling_message . . . . .	45
2.6.9	XM_receive_queuing_message . . . . .	47
2.6.10	XM_send_queuing_message . . . . .	49
2.6.11	XM_write_sampling_message . . . . .	51
2.7	Memory Management . . . . .	53
2.7.1	XM_memory_copy . . . . .	53
2.7.2	XM_set_cache_state . . . . .	55
2.8	Health Monitor Management . . . . .	56
2.8.1	XM_hm_raise_event . . . . .	56
2.8.2	XM_hm_read . . . . .	57
2.8.3	XM_hm_status . . . . .	58
2.9	Trace Management . . . . .	59
2.9.1	XM_trace_event . . . . .	59
2.9.2	XM_trace_read . . . . .	60
2.9.3	XM_trace_status . . . . .	61
2.10	Interrupt Management . . . . .	62
2.10.1	XM_clear_irqmask . . . . .	62
2.10.2	XM_clear_irqpend . . . . .	64
2.10.3	XM_raise_ipvi . . . . .	65
2.10.4	XM_set_irqmask . . . . .	66
2.10.5	XM_set_irqpend . . . . .	67
2.11	Miscellaneous . . . . .	68
2.11.1	XM_get_gid_by_name . . . . .	68
2.11.2	XM_write_console . . . . .	69
2.12	ARM specific . . . . .	70
2.12.1	XM_arm_hm_raise_event . . . . .	70
2.12.2	XM_arm_inport . . . . .	71
2.12.3	XM_arm_outport . . . . .	72
<b>3</b>	<b>Library functions</b>	<b>75</b>
3.1	XEF_load_custom_file . . . . .	76
3.2	XEF_load_file . . . . .	77
3.3	XEF_parse_file . . . . .	78
3.4	init_libxm . . . . .	80
	<b>Hypercall Index</b>	<b>81</b>

This page is intentionally left blank.



# 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 underlying 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.

## Caution sign

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



## Support

Fent Innovative Software Solutions (FentISS)  
Camino de Vera s/n  
CP: 46022  
Valencia, Spain

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

# Chapter 1

## Introduction

This chapter describes the data structures and variables provided by XtratuM.

### 1.1 Global elements

#### 1.1.1 Basic types

The API provides a set of basic types:

```
// 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 1.1: xm-src/core/include/arm/arch\_types.h

#### 1.1.2 Extended types

The API provides a set of basic types:

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

---

Listing 1.2: xm-src/core/include/arm/arch\_types.h

### 1.1.3 Return codes

The API provides a return code as result of a service invocation. The return code is a `xm_s32_t` type. The possible values are:

```
#define XM_OK (0)
#define XM_NO_ACTION (-1)
#define XM_UNKNOWN_HYPERCALL (-2)
#define XM_INVALID_PARAM (-3)
#define XM_PERM_ERROR (-4)
#define XM_INVALID_CONFIG (-5)
#define XM_INVALID_MODE (-6)
#define XM_NOT_AVAILABLE (-7)
#define XM_OP_NOT_ALLOWED (-8)
```

Listing 1.3: xm-src/core/include/hypercalls.h

### 1.1.4 Variables

The API provides the following partition variables:

```
XM_PARTITION_SELF : partition identifier
XM_HYPERVISOR_ID : hypervisor identifier
```

### 1.1.5 API and ABI version numbers

The API provides the following partition variables:

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

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

Listing 1.4: xm-src/core/include/hypercalls.h

## 1.2 System management types

### 1.2.1 System types

The API provides the following system status types:

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

Listing 1.5: xm-src/core/include/objects/status.h

## 1.3 Partition management types

### 1.3.1 Partition types

The API provides the following partition status variables:

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

Listing 1.6: xm-src/core/include/objects/status.h

The valid values of state are:

```
#define XM_STATUS_IDLE 0x0
#define XM_STATUS_READY 0x1
```

```
#define XM_STATUS_SUSPENDED 0x2
#define XM_STATUS_HALTED 0x3
```

The API provides the following system status types:

```
typedef struct {
    xm_u32_t resetStatus;
    xm_u32_t resetCounter;
    /* Number of HM events emitted. */
    xm_u64_t noHmEvents; /* [[OPTIONAL]] */
    /* Number of HW interrupts received. */
    xm_u64_t noIrqs; /* [[OPTIONAL]] */
    /* Current major cycle interation. */
    xm_u64_t currentMaf; /* [[OPTIONAL]] */
    /* Total number of system messages: */
    xm_u64_t noSamplingPortMsgsRead; /* [[OPTIONAL]] */
    xm_u64_t noSamplingPortMsgsWritten; /* [[OPTIONAL]] */
    xm_u64_t noQueuingPortMsgsSent; /* [[OPTIONAL]] */
    xm_u64_t noQueuingPortMsgsReceived; /* [[OPTIONAL]] */
    xmTime_t irqLastOccurence[CONFIG_NO_HWIRQS];
} xmSystemStatus_t;
```

Listing 1.7: xm-src/core/include/objects/status.h

### 1.3.2 Partition Control Table (PCT)

The API provides the Partition Control Table (PCT) to partitions. This type is:

```
typedef struct {
    xm_u32_t magic;
    xm_u32_t xmVersion; // XM version
    xm_u32_t xmAbiVersion; // XM's abi version
    xm_u32_t xmApiVersion; // XM's api version
    xm_u32_t resetCounter;
    xm_u32_t resetStatus;
    xm_u32_t cpuKhz;
#define PCT_GET_PARTITION_ID(pct) ((pct)->id&0xff)
#define PCT_GET_VCPU_ID(pct) ((pct)->id>>8)
    xmId_t id;
    xmId_t noVCpus;
#ifdef CONFIG_ARM
    xm_u32_t irqIndex; // irq number raised
#endif /*CONFIG_ARM*/
    // Copy of kthread->ctrl.flags
    xm_u32_t flags;
#define PARTITION_SYSTEM_F (1<<0) // 1:SYSTEM
#define PARTITION_FP_F (1<<1) // Floating point enabled
#define PARTITION_HALTED_F (1<<2) // 1:HALTED
#define PARTITION_SUSPENDED_F (1<<3) // 1:SUSPENDED
#define PARTITION_READY_F (1<<4) // 1:READY
// #define PARTITION_FLUSH_DCACHE_F (1<<5)
// #define PARTITION_FLUSH_ICACHE_F (1<<6)
#define PARTITION_DCACHE_ENABLED_F (1<<7)
```

```

#define PARTITION_ICACHE_ENABLED_F (1<<8)
    xm_u32_t imgStart;
    xm_u32_t hwIrqs[HWIRQS_VECTOR_SIZE]; // Hw interrupts belonging to the
        partition
    xm_s32_t noPhysicalMemAreas;
    xm_s32_t noCommPorts;
    xm_u8_t name[CONFIG_ID_STRING_LENGTH];
    xm_u32_t hwIrqsPend[HWIRQS_VECTOR_SIZE]; // pending hw irqs
    xm_u32_t hwIrqsMask[HWIRQS_VECTOR_SIZE]; // masked hw irqs
    xm_u32_t extIrqsPend; // pending extended irqs
    xm_u32_t extIrqsMask; // masked extended irqs
    struct pctArch arch;
    struct {
        xm_u32_t noSlot:16, releasePoint:1, reserved:15;
        xm_u32_t id;
        xm_u32_t slotDuration;
    } schedInfo;
    xm_u16_t trap2Vector[NO_TRAPS];
    xm_u16_t hwIrq2Vector[CONFIG_NO_HWIRQS];
    xm_u16_t extIrq2Vector[XM_VT_EXT_MAX];
} partitionControlTable_t;

```

Listing 1.8: xm-src/core/include/guest.h

### 1.3.3 Memory Map Table (MMT)

The API provides the Memory Map Table (MMT) type:

```

struct xmPhysicalMemMap {
    xm_s8_t name[CONFIG_ID_STRING_LENGTH];
    xmAddress_t startAddr;
    xmAddress_t mappedAt;
    xmSize_t size;
#define XM_MEM_AREA_UNMAPPED      (1<<0)
#define XM_MEM_AREA_READONLY     (1<<1)
#define XM_MEM_AREA_UNCACHEABLE  (1<<2)
#define XM_MEM_AREA_ROM          (1<<3)
#define XM_MEM_AREA_FLAG0        (1<<4)
#define XM_MEM_AREA_FLAG1        (1<<5)
#define XM_MEM_AREA_FLAG2        (1<<6)
#define XM_MEM_AREA_FLAG3        (1<<7)
#define XM_MEM_AREA_TAGGED       (1<<8)
    xm_u32_t flags;
};

```

Listing 1.9: xm-src/core/include/guest.h

## 1.4 Time management variables

### 1.4.1 Time management variables

The API provides two monotonic non-decreasing virtual clock references per partition. These clock references are:

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

Additionally, it provides two virtual interrupts per partition associated to the clock management. These interrupts are:

```
#define XM_VT_EXT_HW_TIMER: associated to the clock reference
    XM_HW_CLOCK
#define XM_VT_EXT_EXEC_TIMER: associated to the clock reference
    XM_EXEC_CLOCK
```

## 1.5 Memory management type

### 1.5.1 Memory management types

The valid values for cache type are:

```
#define XM_DCACHE 0x1
#define XM_ICACHE 0x2
```

Listing 1.10: xm-src/core/include/hypercalls.h

The valid values for cache management operations are:

```
#define XM_ACTIVATE_CACHE 0x1
#define XM_DEACTIVATE_CACHE 0x2
#define XM_FLUSH_CACHE 0x3
```

Listing 1.11: xm-src/core/include/hypercalls.h

## 1.6 Plan management type

### 1.6.1 Plan management type

The API provides the plan status type:

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

Listing 1.12: xm-src/core/include/objects/status.h



## 1.7 Inter-partition communication types

### 1.7.1 Sampling ports

The API provides the sampling port status type:

```
typedef struct {
    xmTime_t timestamp;
    xm_u32_t lastMsgSize;
    xm_u32_t flags;
    ///#define XM_COMM_MSG_VALID 0x1
    #define XM_COMM_PORT_STATUS_B 1
    #define XM_COMM_PORT_STATUS_M 0x6
    #define XM_COMM_EMPTY_PORT 0x0
    #define XM_COMM_CONSUMED_MSG (XM_COMM_PORT_STATUS_B<<1)
    #define XM_COMM_NEW_MSG (XM_COMM_PORT_STATUS_B<<2)
} xmSamplingPortStatus_t;
```

Listing 1.13: xm-src/core/include/objects/commports.h

The API provides the sampling port information type:

```
typedef struct {
    char *__gParam portName;
    #define XM_INFINITE_TIME ((xm_u32_t)-1)
    xmTime_t refreshPeriod; // Refresh period.
    xm_u32_t maxMsgSize; // Max message size.
    xm_u32_t direction;
    xmObjDesc_t objDesc; // object descriptor, valid if the port is opened
} xmSamplingPortInfo_t;
```

Listing 1.14: xm-src/core/include/objects/commports.h

### 1.7.2 Queuing ports

The API provides the queuing port status type:

```
typedef struct {
    xm_u32_t noMsgs; // Current number of messages.
} xmQueuingPortStatus_t;
```

Listing 1.15: xm-src/core/include/objects/commports.h

The API provides the sampling port information type:

```
typedef struct {
    char *__gParam portName;
    xm_u32_t maxMsgSize; // Max message size.
    xm_u32_t maxNoMsgs; // Max number of messages.
    xm_u32_t direction;
    xm_u32_t flags;
    xmObjDesc_t objDesc; // object descriptor, valid if the port is opened
} xmQueuingPortInfo_t;
```

---

Listing 1.16: xm-src/core/include/objects/commports.h

## 1.8 Health Monitor types

### 1.8.1 Health Monitor types

The API provides the HM types:

```

struct xmHmLog {
#define XM_HMLOG_SIGNATURE 0xfecf
    xm_u32_t opCode;
#define HMLOG_OPCODE_EVENT_MASK (0x1fff<<HMLOG_OPCODE_EVENT_BIT)
#define HMLOG_OPCODE_EVENT_BIT 19
    // Bits 18 and 17 free
#define HMLOG_OPCODE_SYS_MASK (0x1<<HMLOG_OPCODE_SYS_BIT)
#define HMLOG_OPCODE_SYS_BIT 16
#define HMLOG_OPCODE_VALID_CPUCTX_MASK (0x1<<
    HMLOG_OPCODE_VALID_CPUCTX_BIT)
#define HMLOG_OPCODE_VALID_CPUCTX_BIT 15
#define HMLOG_OPCODE_MODID_MASK (0x7f<<HMLOG_OPCODE_MODID_BIT)
#define HMLOG_OPCODE_MODID_BIT 8
#define HMLOG_OPCODE_PARTID_MASK (0xff<<HMLOG_OPCODE_PARTID_BIT)
#define HMLOG_OPCODE_PARTID_BIT 0
    xmTime_t timestamp;
    union {
#define XM_HMLOG_PAYLOAD_LENGTH 4
        struct hmCpuCtxt cpuCtxt;
        xmWord_t payload[XM_HMLOG_PAYLOAD_LENGTH];
    };
} __PACKED;

typedef struct xmHmLog xmHmLog_t;

```

Listing 1.17: xm-src/core/include/objects/hm.h

The status type is defined as:

```

typedef struct {
    xm_s32_t noEvents;
} xmHmStatus_t;

```

Listing 1.18: xm-src/core/include/objects/hm.h

### 1.8.2 HM actions

The API/OS shall provide the following actions definitions:

```

#define XM_HM_AC_IGNORE 0
#define XM_HM_AC_PARTITION_COLD_RESET 1
#define XM_HM_AC_PARTITION_WARM_RESET 2

```

```
#define XM_HM_AC_HYPERVISOR_COLD_RESET 3
#define XM_HM_AC_HYPERVISOR_WARM_RESET 4
#define XM_HM_AC_SUSPEND 5
#define XM_HM_AC_HALT 6
#define XM_HM_AC_PROPAGATE 7
#define XM_HM_AC_SWITCH_TO_MAINTENANCE 8
```

Listing 1.19: xm-src/core/include/xmconf.h

### 1.8.3 HM events

The API/OS shall provide the following symbols for Health Monitor events triggered by XtratuM:

```
#define XM_HM_EV_INTERNAL_ERROR 0
#define XM_HM_EV_UNEXPECTED_TRAP 1
#define XM_HM_EV_PARTITION_UNRECOVERABLE 2
#define XM_HM_EV_PARTITION_ERROR 3
#define XM_HM_EV_PARTITION_INTEGRITY 4
#define XM_HM_EV_MEM_PROTECTION 5
#define XM_HM_EV_OVERRUN 6
#define XM_HM_EV_SCHED_ERROR 7
#define XM_HM_EV_WATCHDOG_TIMER 8
#define XM_HM_EV_INCOMPATIBLE_INTERFACE 9

#define XM_HM_MAX_GENERIC_EVENTS (XM_HM_EV_INCOMPATIBLE_INTERFACE+1)
```

Listing 1.20: xm-src/core/include/xmconf.h

Additionally ARM CORTEX-A9architecture defines some extra events:

```
#define XM_HM_EV_ARM_UNDEF_INSTR (XM_HM_MAX_GENERIC_EVENTS+0)
#define XM_HM_EV_ARM_PREFETCH_ABORT (XM_HM_MAX_GENERIC_EVENTS+1)
#define XM_HM_EV_ARM_DATA_ABORT (XM_HM_MAX_GENERIC_EVENTS+2)
#define XM_HM_EV_ARM_DATA_ALIGNMENT_FAULT (XM_HM_MAX_GENERIC_EVENTS+3)
#define XM_HM_EV_ARM_DATA_BACKGROUND_FAULT (XM_HM_MAX_GENERIC_EVENTS+4)
#define XM_HM_EV_ARM_DATA_PERMISSION_FAULT (XM_HM_MAX_GENERIC_EVENTS+5)
#define XM_HM_EV_ARM_INSTR_ALIGNMENT_FAULT (XM_HM_MAX_GENERIC_EVENTS+6)
#define XM_HM_EV_ARM_INSTR_BACKGROUND_FAULT (XM_HM_MAX_GENERIC_EVENTS+7)
#define XM_HM_EV_ARM_INSTR_PERMISSION_FAULT (XM_HM_MAX_GENERIC_EVENTS+8)
```

Listing 1.21: xm-src/core/include/arm/xmconf.h

## 1.9 Trace management types

### 1.9.1 Trace types

The API provides the trace event type:

```
struct xmTraceEvent {
    xm_u32_t timestamp; // LSB of time
```

```

    xm_u8_t payload[XM_TRACE_PAYLOAD_LENGTH];
} __PACKED;

typedef struct xmTraceEvent xmTraceEvent_t;

```

Listing 1.22: xm-src/core/include/objects/trace.h

The status type is defined as:

```

typedef struct {
    xm_s32_t noEvents;
} xmTraceStatus_t;

```

Listing 1.23: xm-src/core/include/objects/trace.h

## 1.10 Interrupt management types

### 1.10.1 Interrupt management symbols

In ARM CORTEX-A9 the number of interrupts is very high. For this reason the interrupt masks are represented as a vector instead a word. The label HWIRQS.VECTOR\_SIZE is defined the number of words in the vector.

```

#define CONFIG_NO_HWIRQS 96
#define HWIRQS_VECTOR_SIZE \
    ( (CONFIG_NO_HWIRQS%32) ? \
      (CONFIG_NO_HWIRQS/32)+1 : \
      (CONFIG_NO_HWIRQS/32) )

```

In order to handle the hardware and extended interrupt states (mask, pending) these data structures has been declared:

```

xm_u32_t hwIrqsMask[HWIRQS_VECTOR_SIZE]
xm_u32_t hwIrqsPend[HWIRQS_VECTOR_SIZE]

```

Note that hwIrqs in ARM CORTEX-A9 are maintained for compatibility with other architectures but has no functionality associated.

## Chapter 2

# Hypercalls

## 2.1 System Management

### 2.1.1 XM\_get\_system\_status

**Synopsis:** Get the current status of the system.

**Category:**

System partition service.

**Declaration:**

```
xm_s32_t XM_get_system_status (  
    /*out*/    xmSystemStatus_t *status);
```

**Description:**

Returns the current state of the system into the `status` structure.

The `xmSystemStatus_t` is a data type which contains the fields defined in 1.2.

The fields labeled as `OPTIONAL` will not be updated if the “Enable system/partition status accounting” source configuration parameter is not set. By default it is disabled.

**Return value:**

[XM\_OK]

Successful completion.

[XM\_PERM\_ERROR]

The calling partition is not a system partition.

[XM\_INVALID\_PARAM]

`status` address does not belong to the address space of the calling partition.

**History:**

Introduced in XtratuM 2.2.0.

**Usage examples:**

```
xmSystemStatus_t status;  
...  
XM_get_system_status (&status);
```

**See also:**

`XM_get_partition_status` [Page: 16] .

### 2.1.2 XM\_halt\_system

**Synopsis:** Stop the system.

**Category:**

System partition service.

**Declaration:**

```
xm_s32_t XM_halt_system(void);
```

**Description:**

The board is halted immediately. The whole system is stopped: interrupts are disabled and XtratuM executes an endless loop.

This function shall be used with extreme caution. Only a hardware reset can reboot the system.

This service can only succeed when invoked by a system partition.



**Return value:**

The function does not return if the operation succeeds. In case of error, the return code is:

[XM\_PERM\_ERROR]

The calling partition is not a system partition.

**Rationale:**

This service is provided to allow to stop the system in case of non-recoverable malfunctioning of the hardware.

**Usage examples:**

```
XM_halt_system();  
//This code will never be executed
```

**See also:**

XM\_reset\_system [Page: 14]

### 2.1.3 XM\_reset\_system

**Synopsis:** Reset the system.

**Category:**

System partition service.

**Declaration:**

```
xm_s32_t XM_reset_system(
    /*in */ xm_u32_t resetMode);
```

**Description:**

The system is reset immediately. There are two ways to reset the system depending on the resetMode parameter:

XM\_WARM\_RESET:

XtratuM unconditionally jumps to the XtratuM initialization. All XtratuM data structures are initialised.

XM\_COLD\_RESET:

XtratuM unconditionally jumps back to the resident software entry point. This value shall be specified in the attribute /SystemDescription/ResidentSw/@entryPoint of the XML configuration file. If the ./ResidentSw/@entryPoint attribute is not specified then XtratuM will jump to CONFIG\_RDONLY\_SEC\_ADDR defined in menuconfig.



This function shall be used with extreme caution. The state of the partitions will be lost unless it was saved in permanent memory before the reset.

**Return value:**

The function does not return if the operation succeeds. In case of error, the return code is:

[XM\_PERM\_ERROR]

The calling partition is not a system partition.

[XM\_INVALID\_PARAM]

resetMode is not a valid mode.

**Rationale:**

This service can be used to try to recover from hardware errors.

**Usage examples:**

```
XM_reset_system(XM_COLD_RESET);
//This code is never executed
```

**See also:**

XM\_halt\_system [Page: 13]



## 2.2 Partition Management

### 2.2.1 XM\_get\_partition\_mmap

**Synopsis:** This function returns a pointer to the memory map table (MMT).

**Category:**

Library service

**Declaration:**

```
struct xmPhysicalMemMap *XM_get_partition_mmap (void);
```

**Description:**

XM\_get\_partition\_mmap returns a pointer to the memory map table (MMT) data structure. The MMT is a read-only data structure described in section 1.3

**Return value:** This function always succeeds, returning a pointer to the memory map table (MMT).

**Rationale:**

With this table, the partition can know the amount of memory it is using, and where is it initially mapped. It can also be used to translate virtual to physical addresses.

**Usage examples:**

```
struct xmPhysicalMemMap *xmPM;  
xm_u32_t e;  
...  
xmPM = XM_get_partition_mmap();  
for(e=0; e<XM_params_get_PTC()->noPhysicalMemAreas; e++)  
{  
    printf("%s 0x%x\n", xmPM->name, xmPM->startAddr);  
    xmPM++;  
}
```

**See also:**

XM\_params\_get\_PTC [Page: ??] .

### 2.2.2 XM\_get\_partition\_status

**Synopsis:** Get the current status of a partition.

**Category:**

Standard service / System partition service.

**Declaration:**

```
xm_s32_t XM_get_partition_status (  
    /*in */      xmId_t          id,  
    /*out*/      xmPartitionStatus_t *status);
```

**Description:**

Returns in status a structure containing the current status of the partition referenced by id.

The xmPartitionStatus\_t is a data type containing the fields defined in section 1.3.

The fields labeled as OPTIONAL will not be updated if the “Enable system/partition status accounting” source configuration parameter is not set. It is disabled by default .

**Return value:**

[XM\_OK]

Successful completion.

[XM\_PERM\_ERROR]

The calling partition is not requesting its own status nor is a system partition.

[XM\_INVALID\_PARAM]

id is not a valid partition identifier.

status parameter address does not belong to the address space of the calling partition.

**History:**

Introduced in XtratuM 2.2.0.

**Usage examples:**

```
xmPartitionStatus_t status;  
...  
XM_get_partition_status(XM_PARTITION_SELF, &status);
```

**See also:**

XM\_get\_system\_status [Page: 12] .

### 2.2.3 XM\_halt\_partition

**Synopsis:** Terminates a partition.

**Category:**

Standard service/System partition service.

**Declaration:**

```
xm_s32_t XM_halt_partition (
/*in */    xm_u32_t partitionId);
```

**Description:**

The XM\_halt\_partition() hypercall causes the termination of the partitionId partition. The partition is set to the halt state.

The hypervisor will not schedule the target partition until the partition is reset (XM\_reset\_partition()). If the partition is scheduled by a cyclic scheduler, the time slot allocated to the halted partition is left idle. All the resources allocated to the partition are released: Interrupt lines, I/O ports and communication ports. The RAM memory used by the partition is not deleted.

Only system partitions can halt other partitions different from themselves. Any partition can halt itself. The constant XM\_PARTITION\_SELF represents the calling partition.

The target partition is in charge of copying in a non-volatile medium the information to be saved, if any. Although the RAM memory is not erased when this hypercall is called, depending on how the partition will be reset (system hardware reset, cold partition reset or warm partition reset) the memory may be deleted.

If the target partition was already in halt state, then this hypercall has no effect.

**Return value:**

If the target partition (partitionId) is the same as the calling partition then this function always succeeds, and this hypercall does not return. Otherwise, the return value can be:

[XM\_OK]

The target partition has been successfully halted, or the target partition was already in the halt state.

[XM\_INVALID\_PARAM]

partitionId is not a valid partition identifier.

[XM\_PERM\_ERROR]

The calling partition is not a system partition and the target partition is not the calling partition.

**Usage examples:**

```
...
XM_halt_partition(XM_PARTITION_SELF);
/* This code is never executed unless the partition is restored
   by a system partition*/
```

**See also:**

XM\_reset\_partition [Page: 21] , XM\_suspend\_partition [Page: 26] ,

XM\_resume\_partition [Page: 24] .

### 2.2.4 XM\_idle\_self

**Synopsis:** Idles the execution of the calling partition.

**Category:**

Standard service.

**Declaration:**

```
xm_u32_t XM_idle_self (void);
```

**Description:**

Suspends the execution of the calling partition until a non masked interrupt is received by the partition or until the start of the next scheduling slot, whatever happens first.

**Return value:**

[XM\_OK]

Successful completion.

**Rationale:**

The use of this function improves the overall system performance. Rather than waiting on a busy loop for a trap, the partition can yield the processor to do other activities or to reduce power consumption by lowering the processor frequency (if supported).

The partition developer can use this service to synchronize the execution of the partition with the scheduling plan.

**Usage examples:**

```
#include "guest.h"

#define HW_IRQS 16

...
/* Mask all partition hw interrupts */
for (i=XM_VT_HW_FIRST; i <= HW_VT_HW_LAST; i++) {
    XM_mask_irq(i);
}
/* Mask all partition extended interrupts */
for (i=XM_VT_EXT_FIRST; i <= HW_VT_EXT_LAST; i++) {
    XM_mask_irq(i);
}
/* Unmask the start slot interrupt */
XM_unmask_irq(XM_VT_EXT_CYCLIC_SLOT_START);
XM_enable_irqs();
/* Implementation of a cyclic plan to run partition tasks: */
while (1) {
    TaskA(); TaskB();
    XM_idle_self(); /* Wait for the first slot. */
    TaskC(); TaskB();
    XM_idle_self(); /* Wait for the second slot. */
    TaskD(); TaskE(); TaskB();
    XM_idle_self(); /* Wait for the third slot. */
}
XM_write_console("PANIC: unreachable code\n",25)
```

```
XM_halt_partition(XM_PARTITION_SELF);  
...
```

Note that this code is not robust, and should be improved if used in a product.

### 2.2.5 XM\_params\_get\_PCT

**Synopsis:** Return the address of the PCT.

**Category:**

Library service.

**Declaration:**

```
partitionControlTable_t *XM_params_get_PCT(void);
```

**Description:**

XM\_params\_get\_PCT returns the pointer to the PCT (Partition Control Table) data structure. The PCT is a read-only data structure defined in section 1.3.

**Return value:** This function always succeeds, returning a pointer to the calling partition PCT.

**History:**

Added in version 3.1.2

### 2.2.6 XM\_reset\_partition

**Synopsis:** Reset a partition.

**Category:**

Standard service / System partition service.

**Declaration:**

```
xm_s32_t XM_reset_partition (
    /*in */    xm_u32_t  partitionId,
    /*in */    xmAddress_t ePoint,
    /*in */    xm_u32_t  resetMode
    /*in */    xm_u32_t  status);
```

**Description:**

The partition `partitionId` is reset. The `ePoint` parameter enables to change the default entry point of the partition. Note that `ePoint` becomes the default entry point of the partition. This parameter must be a valid memory are aligned to 4. This value is not mandatory so using an invalid value (e.g.-1) resets the partition to its default entryPoint (obtained from the entryPoint of the partition elf file). There are two ways to reset a partition depending on the `resetMode` value:

**XM\_WARM\_RESET:**

1. The `resetCounter` field of the partition is incremented.
2. The `resetStatus` field of the partition is set to the reset value provided in `status`.
3. The program counter is set to the partition entry point.
4. The partition is set to normal/ready state.

**XM\_COLD\_RESET:**

1. All communication ports are closed.
2. The PCT data structure is initialised in the partition memory space.
3. The `resetCounter` field of the partition is set to zero.
4. The `resetStatus` field of the partition is set to the reset value provided in `status`.
5. The program counter is set to the partition entry point.
6. The partition is set to normal/ready state.

The partition can use the values of `resetCounter` and `resetStatus` to perform different actions after the reset.

Note that the memory of the partition is not modified, i.e, the content of the memory will be the same (except for the PCT) as before the reset.

Only system partitions can reset partitions other than themselves. Any partition can reset itself. The constant `XM_PARTITION_SELF` represents the calling partition.

**Return value:**

If the target partition (`partitionId`) is the calling partition then this function always succeeds, and the hypercall does not return. Otherwise, the return value is:

[XM\_OK]

The target partition has been successfully reset.

[XM\_INVALID\_PARAM]

- `partitionId` is not a valid partition identifier.



- `resetMode` is not a valid reset mode (`XM_COLD_RESET` or `XM_WARM_RESET`).

[`XM_PERM_ERROR`]

The calling partition is not a system partition and the target partition is not the calling partition.

### Usage examples:

```
...  
XM_reset_partition(XM_PARTITION_SELF, -1, XM_COLD_RESET, 0);  
/* This code is never executed */
```

### See also:

`XM_halt_partition` [Page: 17] , `XM_suspend_partition` [Page: 26] ,  
`XM_resume_partition` [Page: 24] , `XM_memory_copy` [Page: 53]



### 2.2.7 XM\_resume\_imm\_partition

**Synopsis:** Resume the execution of a partition immediately.

**Category:**

System partition service.

**Declaration:**

```
xm_s32_t XM_resume_imm_partition (  
    /*in */    xm_u32_t    partitionId);
```

**Description:**

Resumes the execution of the target partition immediately, partitionId. If the target partition is not in suspended state then this function has no effect.

Note: If the time slot of the target partition is active and the partition state is suspended when this service is invoked, the partition shall be resumed immediately. In other cases, this service will take effect in the next slot of the target partition.

All the pending interrupts will be delivered when the partition is resumed.

Only system partitions can invoke this service.

**Return value:**

[XM\_OK]

The target partition has been resumed.

[XM\_INVALID\_PARAM]

partitionId is not a valid partition identifier.

[XM\_PERM\_ERROR]

The calling partition is not system partition.

**Usage examples:**

```
...  
XM_resume_imm_partition(suspendedPartitionId);
```

**See also:**

XM\_suspend\_partition [Page: 26] .

XM\_resume\_partition [Page: 24] .

### 2.2.8 XM\_resume\_partition

**Synopsis:** Resume the execution of a partition.

**Category:**

System partition service.

**Declaration:**

```
xm_s32_t XM_resume_partition (  
    /*in */    xm_u32_t    partitionId);
```

**Description:**

Resumes the execution of partitionId. target partition. If the target partition is not in suspended state then this function has no effect.

All the pending interrupts will be delivered when the partition is resumed.

Only system partitions can invoke this service.

**Return value:**

[XM\_OK]

The target partition has been resumed.

[XM\_INVALID\_PARAM]

partitionId is not a valid partition identifier.

[XM\_PERM\_ERROR]

The calling partition is not a system partition.

**Usage examples:**

```
...  
XM_resume_partition(suspendedPartitionId);
```

**See also:**

XM\_suspend\_partition [Page: 26] .

### 2.2.9 XM\_shutdown\_partition

**Synopsis:** Send a shutdown interrupt to a partition.

**Category:**

Standard service / System partition service.

**Declaration:**

```
xm_s32_t XM_shutdown_partition (
    /*in */    xm_u32_t  partitionId);
```

**Description:**

The XM\_shutdown\_partition() hypercall raises the extended interrupt XM\_VT\_EXT.SHUTDOWN on the target partition.

On receiving a shutdown interrupt (and if the interrupt is unmasked), the target partition should close and terminate the ongoing tasks and finally call the XM\_halt\_partition [Page: 17] hypercall. XtratuM does not control the state of the target partition after a shutdown request.

Only system partitions can invoke this service to shutdown a partition other than themselves. Any partition can shutdown itself. The constant XM\_PARTITION\_SELF represents the calling partition.

If the target partition was in halt state, then this hypercall has no effect.

**Return value:**

If the target partition (partitionId) is the calling partition then this function always succeeds. Otherwise, the return value is:

[XM\_OK]

The shutdown trap has been successfully delivered.

[XM\_INVALID\_PARAM]

partitionId is not a valid partition identifier.

[XM\_PERM\_ERROR]

The calling partition is not a system partition and the target partition is not the calling partition.

**Usage examples:**

```
XM_shutdown_partition(partitionId);
```

**See also:**

XM\_reset\_partition [Page: 21] , XM\_suspend\_partition [Page: 26] ,

XM\_resume\_partition [Page: 24] .

### 2.2.10 XM\_suspend\_partition

**Synopsis:** Suspend the execution of a partition.

**Category:**

Standard service/System partition service.

**Declaration:**

```
xm_s32_t XM_suspend_partition (
/*in */    xm_u32_t  partitionId);
```

**Description:**

Suspends the execution of the partitionId target partition until it is resumed. If the target partition is in suspended state then this hypercall has no effect. The target partition is set in suspended state.

In suspended state, a partition will not be scheduled and interrupts are not delivered. Interrupts raised while in suspended state are left pending. If the partition returns to normal state then pending interrupts will be delivered then.



If a partition is in suspended state for a long period of time, some hardware peripherals may get unattended for an unacceptable amount of time which may cause improper peripheral operation.

Only system partitions can suspend other partitions than themselves. Any partition can suspend itself. The constant XM\_PARTITION\_SELF represents the calling partition.

**Return value:**

[XM\_OK]

The target partition has been suspended or it was already in suspended state.

[XM\_INVALID\_PARAM]

partitionId is not a valid partition identifier.

[XM\_PERM\_ERROR]

The calling partition is not a system partition and the target partition is not the calling partition.

**Usage examples:**

```
...
XM_suspend_partition(XM_PARTITION_SELF);
```

**See also:**

XM\_reset\_partition [Page: 21] , XM\_halt\_partition [Page: 17] ,  
XM\_resume\_partition [Page: 24] .

## 2.3 Multicore Management

### 2.3.1 XM\_get\_vcpuid

**Synopsis:** Returns the *virtual CPU* identifier of the calling partition where the service is invoked.

**Category:**

Standard service.

**Declaration:**

```
xmId_t XM_get_vcpuid (void);
```

**Description:**

Returns the *virtual CPU* for the calling partition.

**Return value:** This function always succeeds. The returned parameter identifies the virtual cpu for the calling partition.

**Rationale:**

The use of this function permits to know the virtual cpu that is used by the code invoking this service for the partition in execution.

### 2.3.2 XM\_halt\_vcpu

**Synopsis:** Halts a *virtual CPU* for the partition invoking this service.

**Category:**

Standard service.

**Declaration:**

```
xm_s32_t XM_halt_vcpu(
    /*in */    xm_u32_t vcpuId);
```

**Description:**

The `XM_halt_vcpu()` hypercall halts the execution of the `vcpu_id` *virtual CPU* for the calling partition. The *virtual CPU* identified in the calling parameter is set to `halt` state.

The halted *virtual CPU* will remain halted until it is reset (`XM_reset_vcpu()`) by the partition. All the execution slots allocated to the *virtual CPU* will not be executed when the *virtual CPU* is halted.

If the target *virtual CPU* was already in `halt` state, then this hypercall has no effect.

**Return value:**

If the target *virtual CPU* (`vcpuId`) is the same as the *virtual CPU* in which the the calling partition invoked this service then this function always succeeds, and this hypercall does not return. Otherwise, the return value can be:

[`XM_OK`]

The target *virtual CPU* for the callign partition has been successfully halted. If the calling partition is running in the *virtual CPU* invoked in the call, the hypercall never returns.

[`XM_INVALID_PARAM`]

`vcpuId` is not a valid *virtual CPU* identifier for the calling partition.

**Usage examples:**

```
...
returnCode = XM_halt_vcpu(XM_VCPU_SELF);
/* This code is never executed unless the partition is restored
   by a system partition*/
...
```

**See also:**

`XM_reset_vcpu` [Page: 29] .

### 2.3.3 XM\_reset\_vcpu

**Synopsis:** Resets a *virtual CPU* for the partition invoking this service.

**Category:**

Standard service.

**Declaration:**

```
xm_s32_t XM_reset_vcpu(  
    /*in */    xm_u32_t    vcpuId,  
    /*in */    xmAddress_t  entryPoint);
```

**Description:**

The *virtual CPU* vcpuId of the calling partition is reset. The *virtual CPU* that is reset will start its execution from the entry point identified in the calling parameters entryPoint. This parameter must be a valid memory address aligned to 4.

The *virtual CPU* of the calling partition is set to normal/ready state.

**Return value:**

The return value of this service is:

[XM\_OK]

The target *virtual CPU* of the calling partition has been successfully reset.

[XM\_INVALID\_PARAM]

entryAddress does not belong to the address space of the calling partition.

vcpuId is not a valid vcpu identifier.

**Usage examples:**

```
...  
XM_reset_vcpu(vcpu, addr1);  
/* This code is never executed */
```

**See also:**

XM\_halt\_vcpu [Page: 28] .

## 2.4 Time Management

### 2.4.1 XM\_get\_time

**Synopsis:** Retrieve the time from the clock specified as a parameter.

**Category:**

Standard service.

**Declaration:**

```
xm_s32_t XM_get_time (
    /*in */    xm_u32_t  clockId,
    /*out*/    xmTime_t  *time);
```

**Description:**

This function obtains the number of **microseconds** elapsed since the last hardware reset as seen by the clock specified in the `clockId` parameter. The retrieved time is stored in the memory address pointed by the `time` parameter.

XtratuM provides two clocks:

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

Listing 2.1: core/include/hypercalls.h

These are monotonic non-decreasing clocks. The resolution is one microsecond, and also the time is represented in microseconds.

Time is represented with an unsigned 64bit integer, which can hold sufficient microseconds to represent more than 290 thousand years.

**Return value:**

[XM\_OK]

Successful completion.

[XM\_INVALID\_PARAM]

`clockId` is a non valid virtual clock.

`time` address does not belong to the calling partition address space.

**Rationale:**

Since the resolution of the clock is 1 microsecond, the same time may be returned if the function is called twice rapidly.

A data type of 64 bits is large enough even to measure the time in nanoseconds, and produce an overflow after a reasonable amount of time.

**Usage examples:**

```
xmTime_t t1,t2,t3;
char msg[100];

XM_get_time(XM_HW_CLOCK, &t1);
XM_get_time(XM_HW_CLOCK, &t2);
do_some_thing();
```



```
XM_get_time(XM_HW_CLOCK, &t3);  
snprintf(msg, 100, "Measured duration: %lld      ", (t3-t2)-(t2-t1  
));  
XM_write_console(msg, 29);
```

**See also:** `XM_set_timer` [Page: 32]

## 2.4.2 XM\_set\_timer

**Synopsis:** Arm a timer.

**Category:**

Standard service.

**Declaration:**

```
xm_s32_t XM_set_timer (
    /*in */    xm_u32_t  clockId,
    /*in */    xmTime_t  absTime,
    /*in */    xmTime_t  interval);
```

**Description:**

If `interval` is zero, then the timer associated with the virtual clock `clockId` is armed **only once** to expire at the absolute instant `absTime`. That is, the timer will expire when the clock reaches the value specified by the `absTime` parameter.

If `interval` is not zero, then the timer will expire **periodically** at absolute times. `absTime n * interval`; where "n" starts in zero and is repeated until the timer is re-armed. The **minimum interval allowed is 50 us**.

If the specified `absTime` time has already passed, the function succeeds and the timer expiration interrupt happens immediately.

Once the timer is armed, the partition will receive a virtual timer interrupt on every timer expiration. It is responsibility of the partition code to install the corresponding interrupt handler.



Clock	Associated extended interrupt
XM_HW_CLOCK	XM_VT_EXT_HW_TIMER
XM_EXEC_CLOCK	XM_VT_EXT_EXEC_TIMER



**There is only one timer per clock.** Therefore, if the timer was already armed when the `XM_set_timer()` function is called, the previous value is reset and the timer is reprogrammed with the new values.

If `absTime` is zero then the timer is disarmed. If at the time of disarming the timer, there were pending timer interrupts, then the interrupts will not be removed, and will be delivered when appropriate. This may happen if `XM_set_timer()` function is called while interrupts are disabled or masked.

If a periodic timer expires several times before the interrupt is received (interrupt line masked, interrupts disabled, or the partition is not ready), then only one interrupt is delivered to the partition.

**Return value:**

[XM\_OK]

Successful completion.

[XM\_INVALID\_PARAM]

- `clockId` is not a valid virtual clock (not a valid clock or the clock cannot be used to arm timers).
- The `absTime` or `interval` parameter has a negative value.
- The interval specified is lower than the minimum `interval` value allowed (50 us.).

**Rationale:**

Internally, timers are managed in **one shot mode**. That is, the hardware timer is not programmed to generate an interrupt periodically, but it is re-programmed to cause the interrupt exactly when the timer closer timer expires.

Although an absolute time point should be a positive number, the time is represented with a signed integer to detect incorrect time values.

**Usage examples:**

```
void ExtIrqHandler(int irqnr) {
    if (irqnr == XM_VT_EXT_HW_TIMER) {
        XM_write_console("Periodic..\n",11);
    } else {
        XM_write_console("Unexpected Irq\n",15);
    }
}

...
xmTime_t Start;
xmTime_t Period = (xmTime_t)10000;

XM_get_time(XM_HW_CLOCK, &Start);
Start += (xmTime_t)1000000;
XM_set_timer(XM_HW_CLOCK, Start, Period);
XM_enable_irqs();
XM_clear_irqmask(0,1<<XM_VT_EXT_HW_TIMER);
```

**See also:** `XM_get_time` [Page: 30]

## 2.5 Plan Management

### 2.5.1 XM\_get\_plan\_status

**Synopsis:** Return information about the status of the scheduling plan.

**Category:**

Standard service.

**Declaration:**

```
xm_s32_t XM_get_plan_status (
/*out*/    xmPlanStatus_t  *status);
```

**Description:**

Returns in `status` the information about the previous, current and next scheduling plans.

The `xmPlanStatus_t` is a data type which contains the fields defined in section 1.6

[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).

**Return value:**

[XM\_OK]

Successful completion.

[XM\_INVALID\_PARAM]

status parameter address does not belong to the address space of the calling partition.

**History:**

Introduced in XtratuM 3.1.2.

**See also:**

`XM_switch_sched_plan` [Page: 36] .

## 2.5.2 XM\_switch\_imm\_sched\_plan

**Synopsis:** Request a plan switch at the end of the current MAF.

**Category:**

System partition service immediately.

**Declaration:**

```
xm_s32_t XM_switch_imm_sched_plan(
/*in */   xm_u32_t newPlanId,
/*out*/   xm_u32_t *currentPlanId);
```

**Description:**

The plan newPlanId is scheduled to be started immediately, no waiting to the end of the current MAF. It returns the current plan identifier.

Note: that this hypercall force an immediate plan switch, instead prepare the system to perform it as in XM\_switch\_sched\_plan.

Note: Changes of operation mode will take effect immediately on all cores. This type of change will interrupt the execution of all the partitions on each core in the system and it will force the beginning of a new MAF for the new scheduling plan.

The plan zero is the initial plan and cannot be called back. Plan zero can only be activated by means of a system reset.

**Return value:**

[XM\_OK]

The operation succeeded.

[XM\_PERM\_ERROR]

The calling partition is not a system partition.

[XM\_INVALID\_PARAM]

currentPlanId is not a valid address or newPlanId is not a valid plan identifier. Valid plan identifiers are those specified in the XM\_CF file, except plan zero.

[XM\_NO\_ACTION]

The newPlanId parameter identifies the current plan.

**History:**

Introduced in XtratuM 3.1.2.

**See also:**

XM\_get\_plan\_status [Page: 34] .

XM\_switch\_imm\_sched\_plan [Page: 35] .

### 2.5.3 XM\_switch\_sched\_plan

**Synopsis:** Request a plan switch at the end of the current MAF.

**Category:**

System partition service.

**Declaration:**

```
xm_s32_t XM_switch_sched_plan(
    /*in */    xm_u32_t  newPlanId,
    /*out*/    xm_u32_t  *currentPlanId);
```

**Description:**

The plan `newPlanId` is scheduled to be started at the end of the current MAF. The current plan identifier is returned in `currentPlanId`.

Note that this hypercall does not force an immediate plan switch, but prepares the system to perform it. If this hypercall is called multiple times, the last call determines the new plan.

The plan zero is the initial plan and cannot be called back. Plan zero can only be activated by means of a system reset.

**Return value:**

[XM\_OK]

Successful completion.

[XM\_PERM\_ERROR]

The calling partition is not a system partition.

[XM\_INVALID\_PARAM]

`currentPlanId` parameter address does not belong to the address space of the calling partition.

`newPlanId` is not a valid plan identifier. Valid plan identifiers are those specified in the `XM.CF` file, except plan zero.

[XM\_NO\_ACTION]

The `newPlanId` parameter identifies the current plan.

**History:**

Introduced in XtratuM 3.1.2.

**See also:**

`XM_get_plan_status` [Page: 34] .

## 2.6 Inter-Partition Communication

### 2.6.1 XM\_create\_queuing\_port

**Synopsis:** Create a queuing port.

**Category:**

Standard service.

**Declaration:**

```
xm_s32_t XM_create_queuing_port (
    /*in */   char      *portName,
    /*in */   xm_u32_t  maxNoMsgs,
    /*in */   xm_u32_t  maxMsgSize,
    /*in */   xm_u32_t  direction);
```

**Description:**

The XM\_create\_queuing\_port() service is used to create a queuing port. Only those queuing ports specified in the configuration file XM\_CF can be created by the partition. Note that all the parameters must match the information contained in the configuration file. New ports cannot be created dynamically.

It is not an error to invoke this hypercall with an already created port. The same port descriptor is returned.

**Return value:**

Upon successful completion, XM\_create\_queuing\_port() returns:

[[port identifier]]

Non-negative integer representing the port descriptor.

[XM\_INVALID\_PARAM]

portName address does not belong to the address space of the calling partition.  
direction contains an invalid value.

[XM\_INVALID\_CONFIG]

- No sampling port of the partition is named portName in the configuration file.
- maxMsgSize is not compatible with the XM\_CF configuration.
- maxNoMsgs not compatible with the XM\_CF configuration.
- direction is not compatible with the XM\_CF configuration.

[XM\_NO\_ACTION]

- The specified port is already open.

**Usage examples:**

```
xm_s32_t portDesc;

portDesc = XM_create_queuing_port ("example",
                                   30, XM_DESTINATION_PORT);
if (portDesc < 0) {
    XM_write_console("Port example could not be created\n",34);
    XM_halt_partition(XM_PARTITION_SELF);
}
```

**See also:**

`XM_send_queuing_message` [Page: [49](#)] , `XM_receive_queuing_message` [Page: [47](#)] ,  
`XM_get_queuing_port_status` [Page: [42](#)] .



## 2.6.2 XM\_create\_sampling\_port

**Synopsis:** Create a sampling port.

**Category:**

Standard service.

**Declaration:**

```
xm_s32_t XM_create_sampling_port (
/*in */   char      *portName,
/*in */   xm_u32_t   maxMsgSize,
/*in */   xm_u32_t   direction,
/*in */   xmTime_t   refreshPeriod);
```

**Description:**

The XM\_create\_sampling\_port() service is used to create a sampling port. Only those sampling ports specified in the configuration file XM\_CF can be created by the partition. Note that all the parameters must match the information contained in the configuration file. New ports cannot be created dynamically.

It is not an error to invoke this hypercall with an already created port. The same port descriptor is returned.

**Return value:**

Upon successful completion, XM\_create\_sampling\_port() returns:

[[port identifier]]

Non-negative integer representing the port descriptor.

[XM\_INVALID\_PARAM]

portName address does not belong to the address space of the calling partition.  
direction contains an invalid value.

[XM\_INVALID\_CONFIG]

No sampling port of the partition is named portName in the configuration file.  
maxMsgSize is out of range or not compatible with the XM\_CF configuration.  
direction is not compatible with the XM\_CF configuration.  
refreshPeriod is not compatible with the XM\_CF configuration.

[XM\_NO\_ACTION]

The specified port is already open.

**Usage examples:**

```
xm_s32_t portDesc;

portDesc = XM_create_sampling_port ("sample",
                                     30, XM_DESTINATION_PORT);
if (portDesc < 0) {
    XM_write_console("Port sample could not be created\n",34);
    XM_halt_partition(XM_PARTITION_SELF);
}
```

**See also:**

XM\_write\_sampling\_message [Page: 51] , XM\_read\_sampling\_message [Page: 45] ,  
XM\_get\_sampling\_port\_status [Page: 44] .

### 2.6.3 XM\_get\_commport\_bitmap

**Synopsis:** Return the address of the communication port bitmap.

**Category:**

Library service.

**Declaration:**

```
xmWord_t *XM_get_commport_bitmap(void);
```

**Description:**

XM\_get\_commport\_bitmap returns a pointer to the communication port bitmap that indicates the status of each port.

**Return value:** This function always succeeds, returning a pointer to the communication port bitmap.

### 2.6.4 XM\_get\_queuing\_port\_info

**Synopsis:** Get the info of a queuing port from the port name.

**Category:**

Standard service.

**Declaration:**

```
xm_s32_t XM_get_queuing_port_info (  
    /*in */    char          *portName,  
    /*out*/    xmQueuingPortInfo_t *info);
```

**Description:**

The XM\_get\_queuing\_port\_info() service is used to get the queuing port static information from the port name. This information includes the identifier of the port. Only those queuing ports specified by the partition in the configuration file XM.CF can be requested by this service. This service fills the structure portInfo, with the fields defined in section 1.7

**Return value:**

The return values are:

[XM\_OK]

Successful completion.

[XM\_INVALID\_PARAM]

portName does not identify an existing queuing port specified by the requesting partition in the XM.CF

portName address does not belong to the address space of the calling partition.

info address does not belong to the address space of the calling partition.

**Usage examples:**

```
#define PORT_NAME "QPortReader"  
xmQueuingPortInfo_t info;  
...  
XM_get_queuing_port_info (PORT_NAME, &info);
```

**See also:**

XM\_send\_queuing\_message [Page: 49] , XM\_receive\_queuing\_message [Page: 47] ,

XM\_create\_queuing\_port [Page: 37] , XM\_get\_queuing\_port\_status [Page: 42] .

### 2.6.5 XM\_get\_queuing\_port\_status

**Synopsis:** Get the status of a queuing port.

**Category:**

Standard service.

**Declaration:**

```
xm_s32_t XM_get_queuing_port_status (  
    /*in */    xm_u32_t      portDesc,  
    /*out*/    xmQueuingPortStatus_t *status);
```

**Description:**

This function fills the structure `status` with the fields defined in section 1.7.

**Return value:**

The return values are:

[XM\_OK]

Successful completion.

[XM\_INVALID\_PARAM]

`portDesc` does not identify a valid queuing port.

`status` address does not belong to the address space of the calling partition.

The port is not open.

**Usage examples:**

```
xmQueuingPortStatus_t portStatus;  
...  
xm_u32_t portDesc = XM_create_queuing_port(portName, maxNoMsgs,  
                                           maxMsgSize, XM_PORT_DESTINATION);  
...  
XM_queuing_port_status (portDesc, &portStatus);
```

**See also:**

`XM_send_queuing_message` [Page: 49] , `XM_receive_queuing_message` [Page: 47] ,

`XM_create_queuing_port` [Page: 37] .

### 2.6.6 XM\_get\_sampling\_port\_info

**Synopsis:** Get the info of a sampling port from the port name.

**Category:**

Standard service.

**Declaration:**

```
xm_s32_t XM_get_sampling_port_info (  
    /*in */   char          *portName,  
    /*out*/   xmSamplingPortInfo_t *info);
```

**Description:**

The XM\_get\_sampling\_port\_info() service is used to get the sampling port static information from the port name. This information includes the identifier of the port. Only those sampling ports specified by the partition in the configuration file XM.CF can be requested by this service. This service fills the structure portInfo, with the fields defined in section 1.7

**Return value:**

[XM\_OK]

Successful completion.

[XM\_INVALID\_PARAM]

portName does not identify an existing sampling port specified by the requesting partition in the XM.CF

portName address does not belong to the address space of the calling partition.

info address does not belong to the address space of the calling partition.

**Usage examples:**

```
#define PORT_NAME "QPortReader"  
xmSamplingPortInfo_t info;  
...  
XM_get_sampling_port_info (PORT_NAME, &info);
```

**See also:**

XM\_write\_sampling\_message [Page: 51] , XM\_read\_sampling\_message [Page: 45] ,

XM\_create\_sampling\_port [Page: 39] , XM\_get\_sampling\_port\_status [Page: 44] .

### 2.6.7 XM\_get\_sampling\_port\_status

**Synopsis:** Get the status of a sampling port.

**Category:**

Standard service.

**Declaration:**

```
xm_s32_t XM_get_sampling_port_status(  
    /*in */    xm_u32_t          portDesc,  
    /*out*/    xmSamplingPortStatus_t *status);
```

**Description:**

This function fills the structure `status`, with the fields defined in section 1.7.

**Return value:**

[XM\_OK]

Successful completion.

[XM\_INVALID\_PARAM]

`portDesc` does not identify a valid sampling port.

`status` address does not belong to the address space of the calling partition.

The port is not open.

[XM\_NOT\_AVAILABLE]

This code is returned by multicore configurations with SMP support when the sampling channel is spinlocked, in order to provide a non-blocking service.

**Usage examples:**

```
xmSamplingPortStatus_t portStatus;  
xm_u32_t portDesc = XM_create_sampling_port(portName,  
                                             maxMsgSize, XM_PORT_DESTINATION  
                                             );  
...  
XM_get_sampling_port_status (portDesc, &portStatus);
```

**See also:**

`XM_write_sampling_message` [Page: 51] , `XM_read_sampling_message` [Page: 45] ,

`XM_create_sampling_port` [Page: 39]

### 2.6.8 XM\_read\_sampling\_message

**Synopsis:** Reads a message from the specified sampling port.

**Category:**

Standard service.

**Declaration:**

```
xm_s32_t XM_read_sampling_message (
/*in */   xm_s32_t portDesc,
/*out*/   void      *msgPtr,
/*in */   xm_u32_t msgSize,
/*out*/   xm_u32_t *flags);
```

**Description:**

The XM\_read\_sampling\_message() service is used to read a message from the specified sampling port. If succeed, at most maxSize bytes are copied into the buffer pointed by msgPtr.

If flags is not a null pointer, then the validity bit (XM\_MSG\_VALID) is set accordingly: the bit is set if the age of the read message is consistent with the @validPeriod optional attribute of the channel. Otherwise the bit is reset. Note that the message is considered to be correct, even if the XM\_MSG\_VALID bit is reset.

**Return value:**

When successful, the service returns the minimum between the length of the received message and the maxSize parameter (i.e. number of bytes read). If the port has not been written yet, them the function returns XM\_NO\_ACTION. In case of error, one of the following negative values is returned:

[XM\_INVALID\_PARAM]

portDesc does not identify a valid sampling port.

The specified sampling port is not configured as XM\_DESTINATION\_PORT.

The flags parameter address does not belong to the address space of the calling partition.

The msgPtr parameter address does not belong to the address space of the calling partition.

The port is not open.

[XM\_INVALID\_CONFIG]

The msgSize is not compatible with the configuration of the sampling channel.

The value of msgSize is zero.

The port is not connected to a sampling channel in the configuration file.

[XM\_NO\_ACTION]

The portDesc port is empty.

**Optimization:**

If the buffer is aligned to 8 bytes, then the copy operation is performed faster. Also, the copy operation is more efficient if the length of the message is multiple of 8 bytes.

**Usage examples:**

```
xm_s32_t portDesc, ret_code;
char message[30];
xm_u32_t flags;

portDesc = XM_create_sampling_port ("sample",
```

```
                                0, XM_PORT_DESTINATION);  
if (portDesc < 0) {  
    XM_halt_partition(XM_PARTITION_SELF);  
}  
ret_code = XM_read_sampling_message (portDesc, message, 30, &flags);  
  
if (ret_code < 0) {  
    XM_write_console("Error reading sampling port\n",28);  
    return;  
}  
if (! (flags & XM_MSG_VALID)) {  
    XM_write_console("The message is not valid!\n",27);  
    return;  
}  
message[29]=0x0; /* For safety */  
XM_write_console(message,ret_code);
```

**See also:**

XM\_create\_sampling\_port [Page: 39] , XM\_write\_sampling\_message [Page: 51] ,  
XM\_get\_sampling\_port\_status [Page: 44] .



### 2.6.9 XM\_receive\_queuing\_message

**Synopsis:** Receive a message from the specified queuing port.

**Category:**

Standard service.

**Declaration:**

```
xm_s32_t XM_receive_queuing_message (
/*in */   xmObjDesc_t portDesc,
/*out*/   void          *msgPtr,
/*in */   xm_u32_t      msgSize);
```

**Description:**

If the port channel is not empty, then the oldest message is retrieved from the channel. At most `maxSize` bytes are copied into the buffer pointed by `msgPtr`. The actual number of bytes received is returned by the call in case of successful completion.

If the hypercall succeeds, then the message is removed from the XtratuM channel. Note that the message is consumed (considered as correctly read) even if the receiving partition only reads the message partially (i.e. the parameter `msgSize` is smaller than the actual size of the message in the channel).

**Return value:**

In case of success, the minimum between the length of the received message and the `msgSize` parameter is returned. On error, one of the following negative values is returned:

[XM\_INVALID\_PARAM]

portDesc does not identify a valid queuing port.

The specified queuing port is not configured as XM\_DESTINATION\_PORT.

The port is not open.

The `msgPtr` parameter address does not belong to the address space of the calling partition.

[XM\_INVALID\_CONFIG]

The `msgSize` is not compatible with the configuration of the queuing channel.

The value of `msgSize` is zero.

The port is not connected to a queuing channel in the configuration file.

[XM\_NOT\_AVAILABLE]

The queuing channel is empty.

**Optimization:**

If the buffer is aligned to 8 bytes, then the copy operation is performed faster. Also, the copy operation is more efficient if the length of the message is multiple of 8 bytes.

**Usage examples:**

```
xm_s32_t portDesc, ret_code;
char message[30];
xm_u32_t flags;

portDesc = XM_create_queuing_port ("queue", 4, 30,
XM_PORT_DESTINATION);
if (portDesc < 0) {
    XM_halt_partition(XM_PARTITION_SELF);
```

```
}
ret_code = XM_receive_queuing_message (portDesc, message, 30, &flags
);
if (ret_code < 0) {
    XM_write_console("Error reading sampling port\n",28);
    return;
}
if (! (flags & XM_MSG_VALID)) {
    XM_write_console("The message is not valid!\n",27);
    return;
}
message[29]=0x0; /* For safety */
XM_write_console(message,ret_code);
```

**See also:**

[XM\\_create\\_queuing\\_port \[Page: 37\]](#) , [XM\\_send\\_queuing\\_message \[Page: 49\]](#) ,  
[XM\\_get\\_queuing\\_port\\_status \[Page: 42\]](#) , [XM\\_get\\_queuing\\_port\\_status \[Page: 42\]](#) .

### 2.6.10 XM\_send\_queuing\_message

**Synopsis:** Send a message in the specified queuing port.

**Category:**

Standard service.

**Declaration:**

```
xm_s32_t XM_send_queuing_message (
    /*in */   xm_s32_t   portDesc,
    /*in */   void       *msgPtr,
    /*in */   xm_u32_t   msgSize);
```

**Description:**

The message is written into the XtratuM internal channel of the port, if enough space is available. Otherwise, the operation fails.

**Return value:**

[XM\_OK]

The message has been successfully written into the port.

[XM\_INVALID\_PARAM]

portDesc does not identify a valid queuing port

The specified queuing port is not configured as XM\_SOURCE\_PORT.

The port is not open.

The msgPtr parameter address does not belong to the address space of the calling partition.

[XM\_INVALID\_CONFIG]

The msgSize is not compatible with the configuration of the queuing channel.

The value of msgSize is zero.

The port is not connected to a queuing channel in the configuration file.

[XM\_NOT\_AVAILABLE]

Insufficient space in the queuing channel.

**Optimization:**

If the buffer is aligned to 8 bytes, then the copy operation is performed faster. Also, the copy operation is more efficient if the length of the message is multiple of 8 bytes.

**Usage examples:**

```
xm_s32_t portDesc, ret_code;
char message="This is a dummy message";
xm_u32_t flags;

portDesc = XM_create_queuing_port ("queue", 4, 30,
    XM_PORT_DESTINATION);
if (portDesc < 0) {
    XM_halt_partition(XM_PARTITION_SELF);
}
ret_code = XM_write_queuing_message (portDesc, message, 24, &flags);
if (ret_code < 0) {
    XM_write_console("Error writing sampling port\n",28);
    return;
}
```

**See also:**

`XM_create_queuing_port` [Page: [37](#)] , `XM_receive_queuing_message` [Page: [47](#)] ,  
`XM_get_queuing_port_status` [Page: [42](#)] , `XM_get_queuing_port_status` [Page: [42](#)] .

### 2.6.11 XM\_write\_sampling\_message

**Synopsis:** Writes a message in the specified sampling port.

**Category:**

Standard service.

**Declaration:**

```
xm_s32_t XM_write_sampling_message (
    /*in */    xm_s32_t  portDesc,
    /*in */    void      *msgPtr,
    /*in */    xm_u32_t  msgSize);
```

**Description:**

The message is automatically copied into the internal XtratuM buffer of the channel. The message will overwrite any previous message in the sampling channel.

If the `validPeriod` parameter is specified in the configuration file, then a timestamp is attached to the message when it is copied to the channel to determine its validity.

**Return value:**

[XM\_OK]

The message has been successfully written into the channel's internal buffer.

[XM\_INVALID\_PARAM]

`portDesc` does not identify a valid sampling port.

The specified sampling port is not configured as `XM_SOURCE_PORT`.

The `msgPtr` parameter address does not belong to the address space of the calling partition.

The port is not open.

[XM\_INVALID\_CONFIG]

The `msgSize` is not compatible with the configuration of the sampling channel.

The value of `msgSize` is zero.

The port is not connected to a sampling channel in the configuration file.

**Optimization:**

If the buffer is aligned to 8 bytes, then the copy operation is performed faster. Also, the copy operation is more efficient if the length of the message is multiple of 8 bytes.

**Usage examples:**

```
xm_s32_t portDesc, retCode;

char message[] = "This is a sample message";

portDesc = XM_create_sampling_port ("sample",
                                    30, XM_PORT_DESTINATION);
if (portDesc < 0) XM_halt_partition(XM_PARTITION_SELF);
retCode = XM_write_sampling_message (portDesc,
                                    message, strlen(message));
if (retCode != XM_OK)
    XM_halt_partition(XM_PARTITION_SELF);
```

**See also:**

`XM_create_sampling_port` [Page: 39] , `XM_read_sampling_message` [Page: 45] ,  
`XM_get_sampling_port_status` [Page: 44] .

## 2.7 Memory Management

### 2.7.1 XM\_memory\_copy

**Synopsis:** Copy a memory area of a specified size from a source to a destination memory area.

**Category:**

Standard service / System partition service.

**Declaration:**

```

xm_s32_t XM_memory_copy (
/*in */    xmId_t    dstId,
/*in */    xm_u32_t  dstAddr,
/*in */    xmId_t    srcId,
/*in */    xm_u32_t  srcAddr,
/*in */    xm_u32_t  size);

```

**Description:**

This function copies data from/to address spaces. The addresses (source and/or destination) can point to memory or to mapped I/O registers.

This function copies *size* bytes from the area pointed by *srcAddr*, located in the address space of *srcId* partition, to the address *dstAddr* in the memory space of *dstId* partition.

The following considerations shall be taken into account:

- The source and destination areas shall not overlap to avoid data corruption.
- When copying from memory to memory, for efficiency reasons, both areas shall be 8 bytes aligned.
- Since this function allows to copy large blocks of memory, precautions must be taken to avoid breaking temporal isolation.
- If the source or destination addresses do not belong to the space of a partition (for example, ROM areas) then the `XM_HYPERVISOR_ID` shall be used.  
Only system partitions are allowed to perform a copy for/to address spaces other than its own (i.e. other partitions or system space).
- If the source or/and destination are mapped I/O registers, then the I/O mapped addresses shall be word (4 bytes) aligned.
- Note that the *size* parameter indicates the number of **bytes** to be copied, and an I/O register is 4 bytes.
- Only memory mapped I/O registers fully allocated to the partition (that is, using the *Range* element in the `XM.CF` configuration file) can be addressed by the `XM_memory_copy` function.

**Return value:**

[`XM_OK`]

Successful completion.

[`XM_INVALID_PARAM`]

- *dstId* or *srcId* are not valid partition identifiers.
- The destination area does not belong to the *dstId* partition, or the source area does not belong to the *srcId* partition.
- The destination is a read-only area.

[XM\_PERM\_ERROR]

The calling partition is not a system partition, and not both `srcId` and `dstId` refer to the calling partition.

### Usage examples:

```
xm_u32_t i=10;
xm_u32_t j;
...
XM_memory_copy(XM_PARTITION_SELF,&i,XM_PARTITION_SELF,&j,sizeof
(xm_u32_t));
```



### 2.7.2 XM\_set\_cache\_state

**Synopsis:** Perform a cache operation.

**Category:**

Standard service.

**Declaration:**

```
xm_s32_t XM_set_cache_state(  
    /*in */    xm_u32_t  cacheType,  
    /*in */    xm_u32_t  cacheOperation);
```

**Description:**

Perform the operation specified by cacheOperation on cacheType.

cacheType can be any of the values specified in 1.5: XM\_DCACHE and XM\_ICACHE.

cacheOperation can be any of the values specified in 1.5: XM\_ACTIVATE\_CACHE, XM\_DEACTIVATE\_CACHE, XM\_FLUSH\_CACHE.

**Return value:**

[XM\_OK]

The operation succeeded.

[XM\_INVALID\_PARAM]

cacheType is not a valid cache type or cacheOperation is not a valid cache operation.

[XM\_INVALID\_CONFIG]

Cache is not enabled in the XM.CF.

**History:**

Added in version 3.3.1

## 2.8 Health Monitor Management

### 2.8.1 XM\_hm\_raise\_event

**Synopsis:** Raises a partition HM event.

**Category:**

Library service.

**Declaration:**

```
xm_s32_t XM_hm_raise_event(  
/* in */   xm_u32_t event);
```

**Description:**

This hypercall allows to raise an error detected by a partition.

If no log entry is available, the action associated to the application error is executed but the error is not logged.

**Return value:**

[XM\_OK]

The operation succeeded.

[XM\_INVALID\_PARAM]

The parameter event is not a valid application error.

**History:**

Introduced in XtratuM 3.9.0.

**See also:**

XM\_hm\_read [Page: 57] , XM\_hm\_status [Page: 58] , XM\_arm\_hm\_raise\_event [Page: 70] .

## 2.8.2 XM\_hm\_read

**Synopsis:** Retrieve health monitoring log entries.

**Category:**

System partition service.

**Declaration:**

```
xm_s32_t XM_hm_read (
/* in */    xmHmLog_t  *hmLogPtr,
/* in */    xm_s32_t   noLogs);
```

**Description:**

This hypercall attempts to read a maximum of noLogs health monitoring log entries into the array pointed by hmLogPtr. The total number of logs retrieved is returned.

Note that this operation is destructive. That is, once read, log entries are removed from the XM internal buffer.

Every health monitoring log entry is a data xmHmLog\_t structure (see hm-types [Page: 8] ).

**Return value:**

[Positive value or 0.]

Number of logs retrieved. If 0 is returned, it means that there were no HM log entries.

[XM\_PERM\_ERROR]

The calling partition is not a system partition.

[XM\_INVALID\_PARAM]

The hmLogPtr parameter address does not belong to the address space of the calling prtion.  
noLogs is equal to 0.

**History:**

Introduced in XtratuM 2.2.0.

**Usage examples:**

```
xmHmLog_t hmLogEntry;
...
while (1) {
    XM_idle_self();
    XM_hm_read(&hmLogEntry, 1);
    ProcessHmEntry(&hmLogEntry);
}
```

**See also:**

XM\_hm\_raise\_event [Page: 56] , XM\_arm\_hm\_raise\_event [Page: 70] , XM\_hm\_status [Page: 58] .

### 2.8.3 XM\_hm\_status

**Synopsis:** Get the status of the health monitoring log stream.

**Category:**

System partition service.

**Declaration:**

```
xm_s32_t XM_hm_status(  
    /*out*/    xmHmStatus_t *hmStatusPtr);
```

**Description:**

This hypercall returns information about the XtratuM health monitoring log stream in the structure `hmStatusPtr`.

This service returns a `xmHmStatus_t` structure, which contains the fields defined in section 1.8.

**Return value:**

[XM\_OK]

The operation succeeded.

[XM\_PERM\_ERROR]

The calling partition is not a system partition.

[XM\_INVALID\_PARAM]

The `hmStatusPtr` parameter address does not belong to the address space of the calling partition.

**History:**

Introduced in XtratuM 2.2.0.

**Usage examples:**

```
xmHmLogStatus_t hmLogStatus;  
...  
XM_hm_status(&hmLogStatus);
```

**See also:**

`XM_hm_read` [Page: 57] , `XM_hm_raise_event` [Page: 56] , `XM_arm_hm_raise_event` [Page: 70]

## 2.9 Trace Management

### 2.9.1 XM\_trace\_event

**Synopsis:** Records a trace entry.

**Category:**

Standard service.

**Declaration:**

```
xm_s32_t XM_trace_event (  
    /*in */    xm_u8_t *trace);
```

**Description:**

This service records the trace event pointed by `trace` into the partition's trace stream.

Each partition has its own trace stream to store the trace events generated by the partition.

An event is an array of `XM_TRACE_PAYLOAD` bytes (8 bytes).

When a partition generates a trace, the event trace is stored jointly with a timestamp. This timestamp enables to know the exact moment when a trace was generated.

The trace parameter address shall be aligned to 8 bytes.

**Return value:**

[`XM_OK`]

Successful completion.

[`XM_INVALID_PARAM`]

The trace address does not belong to the address space of the calling partition.

**History:**

Introduced in XtratuM 2.2.0.

**See also:**

`XM_trace_read` [Page: 60] , `XM_trace_status` [Page: 61] .

## 2.9.2 XM\_trace\_read

**Synopsis:** Read a trace event.

**Category:**

Standard service.

**Declaration:**

```
xm_s32_t XM_trace_read (
    /*in */    xmId_t      id,
    /*out */   xmTraceEvent_t *traceEventPtr,
    /* in */   xm_s32_t    noTraces);
```

**Description:**

Attempts to retrieve noTraces traces from the partition identified by id into the array pointed by traceEventPtr.

The traces generated by the hypervisor can also be retrieved by using XM\_HYPERVISOR\_ID as id.

Note that this operation is destructive. That is, the trace events are removed from the internal stream once read.

A trace event entry is a xmTraceEvent\_t data structure.

It must be noted that, when retrieving traces belonging to internal XtratuM events generated when the "Enable kernel audit events" option is enabled in menuconfig, the partition identifier related to the event can be found in the first position of the payload member of each trace event entry retrieved. Likewise, the event identifier is located in the second position of the payload member.

**Return value:**

[Positive value or 0.]

Number of traces retrieved. If 0 is returned, it means that there were no traces stored.

[XM\_PERM\_ERROR]

The calling partition is not a system partition, and it is not trying to read its own trace log.

[XM\_INVALID\_PARAM]

The traceEventPtr address does not belong to the address space of the calling partition.

id is not a valid identifier.

noTraces is equal to 0.

**History:**

Introduced in XtratuM 2.2.0.

**See also:**

XM\_trace\_event [Page: 59] , XM\_trace\_status [Page: 61] .

### 2.9.3 XM\_trace\_status

**Synopsis:** Get the status of a trace stream.

**Category:**

Standard service.

**Declaration:**

```
xm_s32_t XM_trace_status (  
    /*in */    xmId_t    id,  
    /*out*/    xmTraceStatus_t *traceStatusPtr);
```

**Description:**

This hypercall returns information of the trace stream status in the structure pointed by traceStatusPtr.

This service returns a structure containing the fields defined in section 1.9.

**Return value:**

[XM\_OK]

Successful completion.

[XM\_PERM\_ERROR]

The calling partition is not a system partition.

[XM\_INVALID\_PARAM]

The traceStatusPtr address does not belong to the address space of the calling partition.  
id is not a valid identifier.

**History:**

Introduced in XtratuM 2.2.0.

**See also:**

XM\_trace\_event [Page: 59] , XM\_trace\_read [Page: 60] .

## 2.10 Interrupt Management

### 2.10.1 XM\_clear\_irqmask

**Synopsis:** Unmask interrupts.

**Category:**

Standard service.

**Declaration:**

```
xm_s32_t XM_clear_irqmask(
    /*in */    xm_u32_t *hwIrqsMask,
    /*in */    xm_u32_t extIrqMask);
```

**Description:**

Asynchronous interrupts (extended) can be individually masked. When one of the masked interrupts is raised, XtratuM marks it as pending and does not deliver it to the partition.

This hypercall allows a partition to unmask an interrupt or set of interrupts by clearing the interrupt lines corresponding to the extIrqMask parameter.

The parameter hwIrqsMask is maintained for compatibility but do not have any effect.

The interrupt handlers of the unmasked interrupts will be invoked as soon as the interrupt is delivered to the partition.

Note: Although hwIrqsMask parameter is an input parameter, as result of the number of interrupts (96) and in order to maintain the maximum compatibility with the different platforms, this parameter is a pointer to the interrupt mask that is composed by an array of 32 bits mask. When hwIrqsMask parameter is equal to NULL (0), this parameter is ignored.

Note: The parameter hwIrqsMask is maintained in the hypercall for API compatibility but do not have any effect.

**Return value:**

[XM\_OK]

Successful completion.

**Rationale:**

This function replaces the deprecated XM\_unmask\_irq().

**History:**

Added in version 3.11.0 Modified for version 3.9.11.

**Usage examples:**

```
xm_u32_t hwIrqsMask[HWIRQS_VECTOR_SIZE];

XM_clear_irqmask(NULL, (1<<XM_VT_EXT_HW_TIMER));
//This call doesn't modify the hardware interrupts mask and only
//modifies
//the extended mask in one bit, that is to say, it only unmask the
//HW_TIMER interrupt.

XM_clear_irqmask(hwIrqsMask, 0);
//This call do not have any effect while not unmask any extended irq.
//And the hypercall do not have effect over hardware interrupts.
```



**See also:**

`XM_set_irqmask` [Page: [66](#)] , `XM_set_irqpend` [Page: [67](#)] , `XM_clear_irqpend` [Page: [64](#)] .

## 2.10.2 XM\_clear\_irqpend

**Synopsis:** Clear pending interrupts.

**Category:**

Standard service.

**Declaration:**

```
xm_s32_t XM_clear_irqpend(
    /*in */    xm_u32_t *hwIrqsPend,
    /*in */    xm_u32_t extIrqPend);
```

**Description:**

Asynchronous interrupts (extended) can be globally disabled or individually masked. When one of these interrupts is triggered while being disabled or masked, XtratuM marks it as pending and does not deliver the interrupt to the partition.

This hypercall allows a partition to clear a pending interrupt, preventing XtratuM to deliver it to the partition in the future.

The interrupt lines set in the extIrqPend parameter are cleared.

The parameter hwIrqsPend is maintained for compatibility but do not have any effect.

Note: Although hwIrqsPend parameter is an input parameter, as result of the number of interrupts (96) and in order to maintain the maximum compatibility with the different platforms, this parameter is a pointer to the interrupt mask that is composed by an array of 32 bits mask. When hwIrqsMask parameter is equal to NULL (0), this parameter is ignored.

Note: The parameter hwIrqsMask is maintained in the hypercall for API compatibility but do not have any effect.

**Return value:**

[XM\_OK]

Successful completion.

**History:**

Added in version 3.11.0 Modified for version 3.9.11.

**Usage examples:**

```
xm_u32_t ExtIrqsMask;

XM_clear_irqpend(NULL, (1<<XM_VT_EXT_HW_TIMER));
//This call doesn't modify the hardware pending interrupts and only
//modified the
//extended pending interrupts in one bit, that is to say, it only
//clears the
//HW_TIMER interrupt if it was pending.
```

**See also:**

XM\_set\_irqpend [Page: 67] , XM\_set\_irqmask [Page: 66] , XM\_clear\_irqmask [Page: 62] .

### 2.10.3 XM\_raise\_ipvi

**Synopsis:** Generate an inter-partition virtual interrupt (IPVI) to a partition as specified in the configuration file.

**Category:**

Standard service

**Declaration:**

```
xm_s32_t XM_raise_ipvi (
    /*in */    xm_u8_t no_ipvi);
```

**Description:**

The XM\_raise\_ipvi() hypercall generates a virtual interrupt to one or several partitions as specified in the configuration file (XM\_CF).

The link between the partition that generates the interrupt and the receiver partitions is specified in the channel section of the configuration file.

**Return value:**

[XM\_OK]

Successful completion.

[XM\_INVALID\_PARAM]

no\_ipvi is not a valid virtual interrupt number.

[XM\_NO\_ACTION]

no\_ipvi is not defined or has not a link to other partition in the configuration file.

**Usage examples:**

```
Configuration file:
...
<Channels>
    <Ipvi id="0" sourceId="0" destinationId="1 2" />
    <Ipvi id="1" sourceId="0" destinationId="2" />
</Channels>
...
partition 0 code:
...
xm_s32_t return_code;

return_code = XM_raise_ipvi(XM_VT_EXT_IPVIO);
// partition 1 and 2 will receive the ipvi XM_VT_EXT_IPVIO //
...
rtn=XM_raise_ipvi(XM_VT_EXT_IPVI1);
// partition 2 will receive the ipvi XM_VT_EXT_IPVI1 //
...
XM_halt_partition(XM_PARTITION_SELF);
}
```

### 2.10.4 XM\_set\_irqmask

**Synopsis:** Mask interrupts.

**Category:**

Standard service.

**Declaration:**

```
xm_s32_t XM_set_irqmask(  
    /*in */    xm_u32_t  *hwIrqsMask,  
    /*in */    xm_u32_t  extIrqsMask);
```

**Description:**

This function masks (blocks) extended interrupts. The interrupt lines set in `extIrqsMask` parameters are masked. The parameter `hwIrqsMask` is maintained for compatibility but do not have any effect.

The interrupt handlers of the masked interrupts will not be invoked until the interrupt line is unmasked again.

Note: Although `hwIrqsMask` parameter is an input parameter, as result of the number of interrupts (96) and in order to maintain the maximum compatibility with the different platforms, this parameter is a pointer to the interrupt mask that is composed by an array of 32 bits mask. When `hwIrqsMask` parameter is equal to `NULL` (0), this parameter is ignored.

Note: The parameter `hwIrqsMask` is maintained in the hypercall for API compatibility but do not have any effect.

**Return value:**

[XM\_OK]

Successful completion.

**Rationale:**

This function replaces the deprecated `XM_mask_irq()`.

**History:**

Added in version 3.1.2

**Usage examples:**

```
XM_set_irqmask(NULL, 0xffffffff);  
//This call masks all extended interruptions
```

**See also:**

`XM_clear_irqmask` [Page: 62]

### 2.10.5 XM\_set\_irqpend

**Synopsis:** Set some interrupts as pending.

**Category:**

Standard service.

**Declaration:**

```
xm_s32_t XM_set_irqpend(  
    /*in */    xm_u32_t  *hwIrqsPend,  
    /*in */    xm_u32_t  extIrqsPend);
```

**Description:**

This function simulates interrupt arrivals (i.e, forces hardware and/or extended interrupts). The interrupt lines which are set in the `hwIrqsPend` and `extIrqsPend` parameters are set as pending hardware and extended interrupts respectively.

Asynchronous interrupts must be globally enabled and individually unmasked in order to be delivered to a partition.

Note: Although `hwIrqsPend` parameter is an input parameter, as result of the number of interrupts (96) and in order to maintain the maximum compatibility with the different platforms, this parameter is a pointer to the interrupt mask that is composed by an array of 32 bits mask. When `hwIrqsPend` parameter is equal to `NULL` (0), this parameter is ignored.

Note: The parameter `hwIrqsMask` is maintained in the hypercall for API compatibility but do not have any effect.

**Return value:**

[XM\_OK]

Successful completion.

**History:**

Added in version 3.1.2 Modified for version 3.9.11.

**Usage examples:**

```
XM_set_irq_pend(NULL,0xffffffff);  
//This call leaves the pending HW_IRQ's as they are and sets all  
//EXT_IRQ as pending.
```

**See also:**

`XM_clear_irqpend` [Page: 64] , `XM_set_irqmask` [Page: 66] , `XM_clear_irqmask` [Page: 62] .

## 2.11 Miscellaneous

### 2.11.1 XM\_get\_gid\_by\_name

**Synopsis:** Returns the identifier of an entity defined in the configuration file.

**Category:**

Standard service / System partition service.

**Declaration:**

```
xm_s32_t XM_get_gid_by_name(
/*in */   xm_u8_t   *name,
/*in */   xm_u32_t  entity)
```

**Description:**

Returns the identifier of an entity as defined in the configuration file, obtaining it from the entity name provided as a parameter.

Entity names can be: partition and plan names.

Valid values for entity are: XM\_PARTITION\_NAME and XM\_PLAN\_NAME.

If the entity param is XM\_PARTITION\_NAME, this service is restricted to system partitions. A normal partition can only obtain its partition identifier.

**Return value:**

[Positive return value or 0.]

If the function succeeds, it returns the identifier of the entity.

[XM\_PERM\_ERROR]

The entity provided as a parameter is XM\_PARTITION\_NAME, and the calling partition is not a system partition nor is requesting its own identifier.

[XM\_INVALID\_PARAM]

entity is not a valid entity value.

name parameter address does not belong to the address space of the partition.

[XM\_INVALID\_CONFIG]

There is no entity defined in the configuration file with the provided name.

**History:**

Introduced in XtratuM 3.3.1

### 2.11.2 XM\_write\_console

**Synopsis:** Print a string in the hypervisor console.

**Category:**

Standard service.

**Declaration:**

```
xm_s32_t XM_write_console (  
    /*in */   char      *buffer,  
    /*in */   xm_s32_t  length);
```

**Description:**

Writes up to length bytes from the buffer provided in buffer to the default output console of XtratuM.

The target device where the messages are printed depends on the configuration of XtratuM. During the debugging phase, the XtratuM console is attached to a serial port.

This function is intended only for development and testing purposes, and should not be used in real operation.

The message is completely written to the output device before the function returns.

**Return value:**

[noBytesWritten]

If the function succeeds, it returns the number of bytes written to the default output console of the hypervisor.

[XM\_INVALID\_PARAM]

The buffer parameter address does not belong to the address space of the calling partition.

**Usage examples:**

```
/* Initialization code */  
XM_write_console("Partition 2: Initialization succeed.\n", 37);
```

## 2.12 ARM specific

### 2.12.1 XM\_arm\_hm\_raise\_event

**Synopsis:** Informs XM about a exception to raise a HM event.

**Category:**

Standard service.

**Declaration:**

```
xm_s32_t XM_arm_hm_raise_event(  
/* in */   xm_u32_t hmEvent,  
/* in */   void *   dataEvent);
```

**Description:**

Some exceptions are handled through partition interrupt vector. Therefore it is necessary that the partition inform the hypervisor the occurrence of such failures. This hypercall allows to raise these errors handled by a partition.

If no log entry is available, the action associated to the application error is executed but the error is not logged.

**Return value:**

[XM\_OK]

The operation succeeded.

[XM\_INVALID\_PARAM]

The parameter event is not a valid application error.

**History:**

Introduced in XtratuM 3.9.11.

**See also:**

XM\_hm\_read [Page: 57] , XM\_hm\_status [Page: 58] .



### 2.12.2 XM\_arm\_inport

**Synopsis:** Read from a hardware I/O port.

**Category:**

Standard service.

**Declaration:**

```
xm_s32_t XM_arm_inport(  
    /*in */    xm_u32_t portAddr,  
    /*out*/    xm_u32_t *ptrValue);
```

**Description:**

This service provides direct hardware access to the peripherals. The content of the IO hardware port portAddr is returned in the address pointed by ptrValue.

The configuration file lists the hardware ports that can be used by each partition. In the case of reading from a restricted port, only the mask attribute is applied to filter out the bits not allocated to the partition.

**Return value:**

[XM.OK]

The operation succeeded.

[XM.INVALID.PARAM]

- portAddr is not allocated to this partition,
- portAddr it is not aligned to 4 bytes,
- ptrValue is not a valid partition address.

**Rationale:**

Note that the ARM architecture does not define separate address spaces for memory and peripherals. This hypercall is implemented as a standard load instruction, with the corresponding security checks.

**See also:** XM\_arm\_outport [Page: 72] , XM\_memory\_copy [Page: 53]

### 2.12.3 XM\_arm\_outport

**Synopsis:** Write in a hardware I/O port.

**Category:**

Standard service.

**Declaration:**

```
xm_s32_t XM_arm_outport(
/*in */    xm_u32_t  portAddr,
/*in */    xm_u32_t  value);
```

**Description:**

This service provides direct hardware access to the peripherals. On response to this call, XtratuM writes value into the port portAddr on the native hardware.

The configuration file lists the hardware ports that can be used by each partition. There are two methods to allocate ports to a partition in the configuration file:

**Range of ports:**

A range of ports, **with no restriction**, allocated to the partition. The Range element is used. Example:

```
<Partition ..... >
  <HwResources>
    <IoPorts>
      <Range base="0xf8000008" noPorts="2"/>
    </IoPorts>
  </HwResources>
</Partition>
```

In this example, the ports 0xf8000008 and 0xf800000c are allocated to the partition.

The attributes base and noPorts are mandatory.

**Restricted ports:**

A single port with restrictions on the bits the the partition is allowed to write in. Only those bits that are set in the mask, can be modified by the partition. Note that the port is read before it is finally written. **The read operation shall not cause side effects on the associated peripheral.** Some devices may interpret as interrupt acknowledge to read from a control port. Another source of error happen then the restricted is implemented as an open collector.

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

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

**Return value:**



[XM\_OK]

The value has been successfully written.

[XM\_INVALID\_PARAM]

- portAddr is not allocated to this partition,
- portAddr it is not aligned to 4 bytes,
- the port has been configured as a restricted port and value is out of range.

**Rationale:**

Note that the ARM architecture does not define separate address spaces for memory and peripherals. This hypercall is implemented as a standard store instruction, with the corresponding security checks.

**See also:** XM\_arm\_inport [Page: 71] , XM\_memory\_copy [Page: 53]

This page is intentionally left blank.

## Chapter 3

# Library functions

### 3.1 XEF\_load\_custom\_file

**Synopsis:** This function loads an already parsed XEF file in memory.

**Category:**

Library service.

**Declaration:**

```

5      void *XEF_load_custom_file(
        /*in */ struct xefFile *xefCustomFile,
        /*out */ struct xefCustomFile *customFile);

```

**Description:**

This hypercall loads a xef custom file located in memory according the addresses stored in the XEF file. The XEF had to be already parsed through the XEF\_parse\_file. This file loaded by using the Read/Write operations defined in the paramter xefOps, passed to the XEF\_parse\_file function. This function provides through the customFile parameter the following information:

```

15      struct xefCustomFile {
        xmAddress_t sAddr;
        xmSize_t size;
20      };

```

The start address (sAddr) where the custom file is loaded and it size (size).

**Return value:**

The return value is, on sucess, the partition's entry point. Otherwise, this hypercall returns 0.

**Usage examples:**

```

25      ...
        struct xefOps xefOps;
        struct xefFile xefFile;
        struct xefCustomFile xefCustomFile;

30      xefOps.Write=memcpy;
        xefOps.Read=memcpy;
        xefOps.VAddr2PAddr=0;
        if (XEF_parse_file(sXef, &xefFile, &xefOps)==XEF_OK) {
35          XEF_load_custom_file(&xefFile, &xefCustomFile);
        }

```

**See also:**

XEF\_parse\_file [Page: 78] , XEF\_load\_file [Page: 77]

## 3.2 XEF\_load\_file

**Synopsis:** This function loads an already parsed XEF file in memory.

40

**Category:**

Library service.

**Declaration:**

```
void *XEF_load_file(  
/*in */ struct xefFile *xefFile);
```

45

**Description:**

This hypercall loads a xef file located in memory according the addresses stored in the XEF file. The XEF had to be already parsed through the XEF\_parse\_file. This file loaded by using the Read/Write operations defined in the paramter xefOps, passed to the XEF\_parse\_file function.

50

**Return value:**

The return value is, on sucess, the partition's entry point. Otherwise, this hypercall returns 0.

**Usage examples:**

```
...  
struct xefOps xefOps;  
struct xefFile xefFile;  
  
xefOps.Write=memcpy;  
xefOps.Read=memcpy;  
xefOps.VAddr2PAddr=0;  
if (XEF_parse_file(sXef, &xefFile, &xefOps)==XEF_OK) {  
    XEF_load_file(&xefFile);  
}
```

55

60

65

**See also:**

XEF\_parse\_file [Page: 78] , XEF\_load\_custom\_file [Page: 76]

### 3.3 XEF\_parse\_file

**Synopsis:** Parses a XEF file located in memory.

**Category:**

Library service.

**Declaration:**

```

xm_s32_t XEF_parse_file(
/*in */    xm_u8_t *img,
/*out */   struct xefFile *xefFile,
/*in */    struct xefOps *xefOps);

```

**Description:**

The XEF file pointed by `img` is parsed, storing in `xefFile` the result. `xefFile` is required later on by the hypercalls `XEF_load_file` and `XEF_load_custom_file` to load them. The parameter `xefOps` defines the operations used by these two hypercalls to perform such load. These operations are:

```

struct xefOps {
    void (*Read)(void *dest, const void *src, unsigned long n);
    void (*Write)(void *dest, const void *src, unsigned long n);
    void (*VAddr2PAddr)(void *, xm_s32_t, xmAddress_t, xmAddress_t
        *);
    void *mAreas;
    xm_s32_t noAreas;
};

```

**[Read]**

A pointer to the function used to read from the XEF file. Note that in the case of compressed XEF files, the XEF file is uncompressed before being loaded at its final location. For this case, Read must support to read from the location of the XEF file and write within the scope of code loading the XEF file.

**[Write]**

A pointer to the function used to write the XEF file to its final destination.

**[VAddr2PAddr]**

The XEF file defines a set of addresses (XEF section address) defining the places where the content of the XEF file is loaded, this function allows to translate these addresses in order to perform a relocation of the partition.

**[mAreas]**

Memory areas passed as parameter to the `VAddr2PAddr` function.

**[noAreas]**

Number of memory areas passed as parameter to the `VAddr2PAddr` function.

**Return value:**

**[XEF\_OK]**

The XEF file has been successfully parsed.

**[XEF\_BAD\_SIGNATURE]**

Unexpected XEF signature found in the XEF image.



[XEF\_INCORRECT\_VERSION]

Unexpected XEF version number found.

[XEF\_UNMATCHING\_DIGEST]

Digest function value does not match with stored one.

115

### Usage examples:

```
...
struct xefOps xefOps;
struct xefFile xefFile;
xefOps.Write=memcpy;
xefOps.Read=memcpy;
xefOps.VAddr2PAddr=0;
XEF_parse_file(sXef, &xefFile, &xefOps);
```

120

125

### See also:

XEF\_load\_file [Page: 77] , XEF\_load\_custom\_file [Page: 76]

## 3.4 init\_libxm

**Synopsis:** Performs libXM parameters initialisation.

130 **Declaration:**

```
void init_libxm(partitionControlTable_t *partCtrlTab)
```

**Description:**

135 The `init_libxm` performs initialisation tasks for libXM. It stores a reference to the PCT passed from XM at partition boot time, and calculates the offsets to the partition memory map and the communication ports bitmap.

**Return value:** The operation always succeeds, not returning any value.

# Index

	init_libxm, 80	
140	XEF_load_custom_file, 76	
	XEF_load_file, 77	
	XEF_parse_file, 78	
	XM_arm_hm_raise_event, 70	
	XM_arm_inport, 71	
145	XM_arm_outport, 72	
	XM_clear_irqmask, 62	
	XM_clear_irqpend, 64	
	XM_create_queuing_port, 37	
	XM_create_sampling_port, 39	
150	XM_get_commpport_bitmap, 40	
	XM_get_gid_by_name, 68	
	XM_get_partition_mmap, 15	
	XM_get_partition_status, 16	
	XM_get_plan_status, 34	
155	XM_get_queuing_port_info, 41	
	XM_get_queuing_port_status, 42	
	XM_get_sampling_port_info, 43	
	XM_get_sampling_port_status, 44	
	XM_get_system_status, 12	
160	XM_get_time, 30	
	XM_get_vcpuid, 27	
	XM_halt_partition, 17	
	XM_halt_system, 13	
	XM_halt_vcpu, 28	
165	XM_hm_raise_event, 56	
	XM_hm_read, 57	
	XM_hm_status, 58	
	XM_idle_self, 18	
	XM_memory_copy, 53	
170	XM_params_get_PCT, 20	
	XM_raise_ipvi, 65	
	XM_read_sampling_message, 45	
	XM_receive_queuing_message, 47	
	XM_reset_partition, 21	
175	XM_reset_system, 14	
	XM_reset_vcpu, 29	
	XM_resume_imm_partition, 23	
	XM_resume_partition, 24	
	XM_send_queuing_message, 49	
180	XM_set_cache_state, 55	
	XM_set_irqmask, 66	
	XM_set_irqpend, 67	
	XM_set_timer, 32	
	XM_shutdown_partition, 25	
	XM_suspend_partition, 26	185
	XM_switch_imm_sched_plan, 35	
	XM_switch_sched_plan, 36	
	XM_trace_event, 59	
	XM_trace_read, 60	
	XM_trace_status, 61	190
	XM_write_console, 69	
	XM_write_sampling_message, 51	