

XtratuM Hypervisor for LEON2

Volume 4: Reference Manual

Miguel Masmano
Ismael Ripoll
Alfons Crespo
Patricia Balbastre

October, 2009
Reference: XM-reference-007e

This page is intentionally left blank.

DOCUMENT CONTROL PAGE

TITLE: XtratuM Hypervisor for LEON2 Volume 4: Reference Manual

AUTHOR/S: Miguel Masmano
Ismael Ripoll
Alfons Crespo
Patricia Balbastre

LAST PAGE NUMBER: 64

VERSION OF SOURCE CODE: XtratuM 2.2.2 for LEON2

REFERENCE ID: XM-reference-007e

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.

Copyright © October, 2009 Miguel Masmano, Ismael Ripoll and Alfons Crespo

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

Changes:

Version	Date	Comments
0.1	April, 2009	Initial document.
0.2	May, 2009	[XM-reference-007] First draft release.
0.3	June, 2009	[XM-reference-007b] Updated with the comments from Lionel.
0.4	July, 2009	[XM-reference-007c] Updated with the comments of the 26/6/2009 phone call meeting. <ul style="list-style-type: none"> • XM_memory_copy now checks the parameters. • XM_memory_copy: areas not in partitions can also be copied. • communication facility: <ul style="list-style-type: none"> Zero size messages is an error. Read/receive returns the number of bytes. Write/send returns OK (not the number of bytes)
0.5	July, 2009	[XM-reference-007d] Final version document for XtratuM v2.2 HM processor triggered events clarified (relation between HM event numbers and trap numbers). Health monitoring overview section.

Version	Date	Comments
0.6	October, 2009	<p data-bbox="531 322 1007 351">[XM-reference-007e] for XtratuM v2.2.2.</p> <ul data-bbox="531 353 1366 497" style="list-style-type: none"><li data-bbox="531 353 1366 416">• <code>XM_memory_copy()</code> extended to allow all partitions to copy from/to its own allocated I/O mapped registers. <li data-bbox="531 450 1366 497">• <code>XM_send_queuing_message()</code> and <code>XM_receive_queuing_message()</code> return the error code <code>XM_NOT_</code> when the port is not connected.

Contents

Preface	vii
1 Introduction	1
1.1 XtratuM Overview	2
1.2 Health Monitoring overview	5
2 Hypercalls	9
2.1 XM_create_queuing_port	10
2.2 XM_create_sampling_port	11
2.3 XM_disable_irqs	12
2.4 XM_enable_irqs	13
2.5 XM_get_queuing_port_status	14
2.6 XM_get_sampling_port_status	15
2.7 XM_get_time	16
2.8 XM_halt_partition	17
2.9 XM_halt_system	18
2.10 XM_hm_open	19
2.11 XM_hm_read	20
2.12 XM_hm_seek	21
2.13 XM_hm_status	23
2.14 XM_idle_self	24
2.15 XM_mask_irq	26
2.16 XM_memory_copy	27
2.17 XM_multicall	29
2.18 XM_partition_get_status	30
2.19 XM_read_sampling_message	31
2.20 XM_receive_queuing_message	33
2.21 XM_request_irq	34
2.22 XM_reset_partition	35

2.23 XM_reset_system	37
2.24 XM_resume_partition	38
2.25 XM_send_queuing_message	39
2.26 XM_set_timer	40
2.27 XM_shutdown_partition	42
2.28 XM_sparcv8_atomic_add	43
2.29 XM_sparcv8_atomic_and	44
2.30 XM_sparcv8_atomic_or	45
2.31 XM_sparcv8_flush_regwin	46
2.32 XM_sparcv8_get_flags	47
2.33 XM_sparcv8_inport	48
2.34 XM_sparcv8_iret	49
2.35 XM_sparcv8_outport	50
2.36 XM_sparcv8_set_flags	52
2.37 XM_suspend_partition	53
2.38 XM_system_get_status	54
2.39 XM_trace_event	55
2.40 XM_trace_open	57
2.41 XM_trace_read	58
2.42 XM_trace_seek	59
2.43 XM_trace_status	60
2.44 XM_unmask_irq	61
2.45 XM_write_console	62
2.46 XM_write_register32	63
2.47 XM_write_sampling_message	64

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 LEON2 (Sparc v8) architecture and experience in programming device drivers. It is also advisable to have some knowledge of the ARINC-653 and related standards.

Typographical conventions

The following font conventions are used in this document:

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

Code

Code examples are printed inside a box like this:

```
#define XM_sparcv8_fill_reg_windows(_area) do { \  
    __asm__ __volatile__ ("mov 2, %%o0\n\t" \  
                          "mov %0, %%o1\n\t" \  
                          "ta "XM_FSYSCALL_TRAP_STR"\n\t" \  
                          "mov %%o1, %0":"=r"(_area):"0"(_area)); \  
} while(0) /* libxm macro */
```

Listing 1: Sample code

Caution sign

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



Support

Alfons Crespo
Universidad Politecnica de Valencia
Instituto de Automtica e Informtica Industrial

Camino de vera s/n
CP: 46022
Valencia, Spain

The official XtratuM web site is: <http://www.xtratum.org>

Acknowledgements

This work has been done in the frame of a bilateral contract between CNES and UPV.

Chapter 1

Introduction

1.1 XtratuM Overview

Synopsis: Global data structures and types.

Global data 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: core/include/sparcv8/arch.types.h

```
// Extended types
typedef xm_s64_t xmTime_t;
typedef xm_u32_t xmAddress_t;
typedef xm_u32_t xmSize_t;
typedef xm_s32_t xmSSize_t;
typedef xm_u32_t xmId_t;
```

Listing 1.2: core/include/sparcv8/arch.types.h

API and ABI version numbers:

```
#define XM_ABI_VERSION 2
#define XM_ABI_SUBVERSION 0
#define XM_ABI_REVISION 0

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

Listing 1.3: core/include/hypercalls.h

Error codes:

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

Listing 1.4: core/include/hypercalls.h

Interrupts:

Native hardware interrupts (interrupts generated by the real hardware) are received by the partition in the same way than in the host machine. The first native interrupt causes the trap number

0x11. The last interrupt (15) causes the trap 0x1F. Although the SPARC v8 architecture defines only 15 interrupt numbers (levels), XtratuM reserves 32 for compatibility with other processors.

Extended interrupts are raised in response to XtratuM events. The new interrupts set are in the range from 32 to 63, and causes traps from 256 to 287 respectively.

Below the list of hardware and extended interrupts (terrupts available in the AT697 board and tsim simulator):

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

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

#define XM_VT_EXT_FIRST        (32)
#define XM_VT_EXT_LAST        (63)
#define XM_VT_EXT_MAX         (32)

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

#define XM_VT_EXT_CYCLIC_SLOT_START (8+XM_VT_EXT_FIRST)

#define XM_VT_EXT_MEM_PROTECT  (16+XM_VT_EXT_FIRST)
```

Listing 1.5: core/include/guest.h

The ”_VT_” prefix stands for **Virtual Trap**.

Note that the trap table of a partition shall contain 32 more entries (to accomodate the extended interrupt set) than the original SPARC v8 trap table.

Exceptions:

List of exceptions (traps triggered by the processor):

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

```
#define UNIMPLEMENTED_FLUSH 0x25 // 9
#define WATCHPOINT_DETECTED 0xb // 10
//#define WINDOW_OVERFLOW 0x5
//#define WINDOW_UNDERFLOW 0x6
#define MEM_ADDRESS_NOT_ALIGNED 0x7 // 11
#define FP_EXCEPTION 0x8 // 12
#define CP_EXCEPTION 0x28 // 13
#define DATA_ACCESS_ERROR 0x29 // 14
#define DATA_ACCESS_MMU_MISS 0x2c // 15
#define DATA_ACCESS_EXCEPTION 0x9 // 16
#define TAG_OVERFLOW 0xa // 17
#define DIVISION_BY_ZERO 0x2a // 18
```

Listing 1.6: core/include/sparcv8/irqs.h

Processor exceptions are managed through the health monitoring system.

Note that WINDOW_OVERFLOW and WINDOW_UNDERFLOW exceptions are automatically managed by XtratuM and hidden to the partition.

1.2 Health Monitoring overview

Synopsis: Health Monitoring overview.

Health monitoring events:

There are three groups of hm events:

Processor triggered

The processor traps caused by an incorrect behavior of the processor or the board are managed as health monitoring events.

```
#define XM_HM_EV_WRITE_ERROR (XM_HM_MAX_GENERIC_EVENTS+0)
#define XM_HM_EV_INSTR_ACCESS_MMU_MISS (XM_HM_MAX_GENERIC_EVENTS+1)
#define XM_HM_EV_INSTR_ACCESS_ERROR (XM_HM_MAX_GENERIC_EVENTS+2)
#define XM_HM_EV_REGISTER_HARDWARE_ERROR (XM_HM_MAX_GENERIC_EVENTS+3)
#define XM_HM_EV_INSTR_ACCESS_EXCEPTION (XM_HM_MAX_GENERIC_EVENTS+4)
#define XM_HM_EV_PRIVILEGED_INSTR (XM_HM_MAX_GENERIC_EVENTS+5)
#define XM_HM_EV_ILLEGAL_INSTR (XM_HM_MAX_GENERIC_EVENTS+6)
#define XM_HM_EV_FP_DISABLED (XM_HM_MAX_GENERIC_EVENTS+7)
#define XM_HM_EV_CP_DISABLED (XM_HM_MAX_GENERIC_EVENTS+8)
#define XM_HM_EV_UNIMPLEMENTED_FLUSH (XM_HM_MAX_GENERIC_EVENTS+9)
#define XM_HM_EV_WATCHPOINT_DETECTED (XM_HM_MAX_GENERIC_EVENTS+10)
#define XM_HM_EV_MEM_ADDR_NOT_ALIGNED (XM_HM_MAX_GENERIC_EVENTS+11)
#define XM_HM_EV_FP_EXCEPTION (XM_HM_MAX_GENERIC_EVENTS+12)
#define XM_HM_EV_CP_EXCEPTION (XM_HM_MAX_GENERIC_EVENTS+13)
#define XM_HM_EV_DATA_ACCESS_ERROR (XM_HM_MAX_GENERIC_EVENTS+14)
#define XM_HM_EV_DATA_ACCESS_MMU_MISS (XM_HM_MAX_GENERIC_EVENTS+15)
#define XM_HM_EV_DATA_ACCESS_EXCEPTION (XM_HM_MAX_GENERIC_EVENTS+16)
#define XM_HM_EV_TAG_OVERFLOW (XM_HM_MAX_GENERIC_EVENTS+17)
#define XM_HM_EV_DIVIDE_EXCEPTION (XM_HM_MAX_GENERIC_EVENTS+18)
```

Listing 1.7: core/include/sparcv8/xmconf.h

User triggered

A trace message with the criticality level equal to XM_TRACE_UNRECOVERABLE raises the XM_HM_EV_PARTITION_ERROR event.

XtratuM triggered

```
#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
```

Listing 1.8: core/include/xmconf.h

Default actions:

The following table summarises the list of health monitoring events (first column) and the default action, both when the error occurs at XtratuM and partition scope. For the processor triggered

events, the trap that raised the event is specified in the last column. If this HM event is propagated to the partition, the trap number raised is the same than the one of the native hardware, and not the HM event number. Note that **XM_HM_EV_*** constants do not match the trap numbers.

Event name (XM_HM_EV_)	XtratuM scope			Partition scope			
	Default (XM_HM_AC_)	action	Log	Default (XM_HM_AC_)	action	Log	Trap #
Processor triggered							
_WRITE_ERROR	._SYSTEM_WARM_RESET	Yes	Yes	._PROPAGATE	No	No	0x2B
_INST_ACC_EXCEPTION	._SYSTEM_WARM_RESET	Yes	Yes	._PROPAGATE	Yes	Yes	0x01
_ILLEGAL_INST	._SYSTEM_WARM_RESET	Yes	Yes	._PROPAGATE	No	No	0x02
_PRIVILEGED_INST	._SYSTEM_WARM_RESET	Yes	Yes	._PROPAGATE	Yes	Yes	0x03
_FP_DISABLED	._SYSTEM_WARM_RESET	Yes	Yes	._PROPAGATE	No	No	0x04
_CP_DISABLED	._SYSTEM_WARM_RESET	Yes	Yes	._PROPAGATE	No	No	0x24
_REG_HW_ERROR	._SYSTEM_WARM_RESET	Yes	Yes	._PARTITION_SUSPEND	Yes	Yes	0x20
_MEM_ADDR_NOT_ALIG	._SYSTEM_WARM_RESET	Yes	Yes	._PROPAGATE	No	No	0x07
_FP_EXCEPTION	._SYSTEM_WARM_RESET	Yes	Yes	._PROPAGATE	No	No	0x08
_DATA_ACC_EXCEPTION	._SYSTEM_WARM_RESET	Yes	Yes	._PROPAGATE	Yes	Yes	0x09
_TAG_OVERFLOW	._SYSTEM_WARM_RESET	Yes	Yes	._PROPAGATE	No	No	0x0A
_DIVIDE_EXCEPTION	._SYSTEM_WARM_RESET	Yes	Yes	._PROPAGATE	No	No	0x2A
User triggered							
_PARTITION_ERROR				._PARTITION_HALT	Yes	Yes	
XtratuM triggered							
_INTERNAL_ERROR	._SYSTEM_WARM_RESET	Yes	Yes				
_UNEXPECTED_TRAP	._IGNORE	Yes	Yes				
_OVERRUN	._IGNORE	Yes	Yes				
_SCHED_ERROR	._IGNORE	Yes	Yes				
_PARTITION_UNRECOVERABLE				._PARTITION_HALT	Yes	Yes	
_PARTITION_INTEGRITY				._PARTITION_HALT	Yes	Yes	
_MEM_PROTECTION				._PARTITION_SUSPEND	Yes	Yes	
_INCOMPATIBLE_INTERFACE				._PARTITION_HALT	Yes	Yes	

Table 1.1: Default actions and log flag.

In the case of the hardware detected events, the last column shows the trap number associated with the hm event.

Allowed configuration actions:

Depending on the severity and the type of HM event, some actions are not applicable. The next table defines which actions are allowed to be used with each HM event:

	XM_HM_AC_COLD_RESET	XM_HM_AC_HALT	XM_HM_AC_IGNORE	XM_HM_AC_PROPAGATE	XM_HM_AC_SHUTDOWN	XM_HM_AC_SUSPEND	XM_HM_AC_WARM_RESET
XM_HM_EV_CP_DISABLED	x	x	x	x	x	x	x
XM_HM_EV_CP_EXCEPTION	x	x	x	x	x	x	x
XM_HM_EV_DATA_ACCESS_ERROR	x	x	x	x	x	x	x
XM_HM_EV_DATA_ACCESS_EXCEPTION	x	x	x	x	x	x	x
XM_HM_EV_DATA_ACCESS_MMU_MISS	x	x	x	x	x	x	x
XM_HM_EV_DIVIDE_EXCEPTION	x	x	x	x	x	x	x
XM_HM_EV_FP_DISABLED	x	x	x	x	x	x	x
XM_HM_EV_FP_EXCEPTION	x	x	x	x	x	x	x
XM_HM_EV_ILLEGAL_INSTR	x	x	x	x	x	x	x
XM_HM_EV_INCOMPATIBLE_INTERFACE	x	x	x		x	x	x
XM_HM_EV_INSTR_ACCESS_ERROR	x	x	x	x	x	x	x

	XM_HM_AC_COLD_RESET	XM_HM_AC_HALT	XM_HM_AC_IGNORE	XM_HM_AC_PROPAGATE	XM_HM_AC_SHUTDOWN	XM_HM_AC_SUSPEND	XM_HM_AC_WARM_RESET
XM_HM_EV_INSTR_ACCESS_EXCEPTION	x	x	x	x	x	x	x
XM_HM_EV_INSTR_ACCESS_MMU_MISS	x	x	x	x	x	x	x
XM_HM_EV_INTERNAL_ERROR	x	x	x		x	x	x
XM_HM_EV_MEM_ADDR_NOT_ALIGNED	x	x	x	x	x	x	x
XM_HM_EV_MEM_PROTECTION	x	x	x		x	x	x
XM_HM_EV_OVERRUN	x	x	x		x	x	x
XM_HM_EV_PARTITION_ERROR	x	x	x		x	x	x
XM_HM_EV_PARTITION_INTEGRITY	x	x	x		x	x	x
XM_HM_EV_PARTITION_UNRECOVERABLE	x	x	x		x	x	x
XM_HM_EV_PRIVILEGED_INSTR	x	x	x	x	x	x	x
XM_HM_EV_REGISTER_HARDWARE_ERROR	x	x	x	x	x	x	x
XM_HM_EV_SCHED_ERROR	x	x	x		x	x	x
XM_HM_EV_TAG_OVERFLOW	x	x	x	x	x	x	x
XM_HM_EV_UNEXPECTED_TRAP	x	x	x		x	x	x
XM_HM_EV_UNIMPLEMENTED_FLUSH	x	x	x	x	x	x	x
XM_HM_EV_WATCHDOG_TIMER	x	x	x		x	x	x
XM_HM_EV_WATCHPOINT_DETECTED	x	x	x	x	x	x	x
XM_HM_EV_WRITE_ERROR	x	x	x	x	x	x	x

Table 1.2: Partition scope HM events

	XM_HM_AC_COLD_RESET	XM_HM_AC_HALT	XM_HM_AC_IGNORE	XM_HM_AC_PROPAGATE	XM_HM_AC_SHUTDOWN	XM_HM_AC_SUSPEND	XM_HM_AC_WARM_RESET
XM_HM_EV_CP_DISABLED	x	x	x				x
XM_HM_EV_CP_EXCEPTION	x	x	x				x
XM_HM_EV_DATA_ACCESS_ERROR	x	x	x				x
XM_HM_EV_DATA_ACCESS_EXCEPTION	x	x	x				x
XM_HM_EV_DATA_ACCESS_MMU_MISS	x	x	x				x
XM_HM_EV_DIVIDE_EXCEPTION	x	x	x				x
XM_HM_EV_FP_DISABLED	x	x	x				x
XM_HM_EV_FP_EXCEPTION	x	x	x				x
XM_HM_EV_ILLEGAL_INSTR	x	x	x				x
XM_HM_EV_INCOMPATIBLE_INTERFACE	x	x	x		x	x	x
XM_HM_EV_INSTR_ACCESS_ERROR	x	x	x				x
XM_HM_EV_INSTR_ACCESS_EXCEPTION	x	x	x				x
XM_HM_EV_INSTR_ACCESS_MMU_MISS	x	x	x				x
XM_HM_EV_INTERNAL_ERROR	x	x	x				x
XM_HM_EV_MEM_ADDR_NOT_ALIGNED	x	x	x				x
XM_HM_EV_MEM_PROTECTION	x	x	x				x

	XM_HM_AC_COLD_RESET	XM_HM_AC_HALT	XM_HM_AC_IGNORE	XM_HM_AC_PROPAGATE	XM_HM_AC_SHUTDOWN	XM_HM_AC_SUSPEND	XM_HM_AC_WARM_RESET
XM_HM_EV_OVERRUN	x	x	x				x
XM_HM_EV_PARTITION_ERROR	x	x	x				x
XM_HM_EV_PARTITION_INTEGRITY	x	x	x				x
XM_HM_EV_PARTITION_UNRECOVERABLE	x	x	x				x
XM_HM_EV_PRIVILEGED_INSTR	x	x	x				x
XM_HM_EV_REGISTER_HARDWARE_ERROR	x	x	x				x
XM_HM_EV_SCHED_ERROR	x	x	x				x
XM_HM_EV_TAG_OVERFLOW	x	x	x				x
XM_HM_EV_UNEXPECTED_TRAP	x	x	x				x
XM_HM_EV_UNIMPLEMENTED_FLUSH	x	x	x				x
XM_HM_EV_WATCHDOG_TIMER	x	x	x		x	x	x
XM_HM_EV_WATCHPOINT_DETECTED	x	x	x				x
XM_HM_EV_WRITE_ERROR	x	x	x				x

Table 1.3: System scope HM events

Health monitoring actions:

[XM_HM_AC_IGNORE] No action is performed.

[XM_HM_AC_PARTITION_COLD_RESET] The offending partition is cold reset.

[XM_HM_AC_PARTITION_WARM_RESET] The offending partition is warm reset.

[XM_HM_AC_PARTITION_SUSPEND] The offending partition is suspended.

[XM_HM_AC_PARTITION_HALT] The offending partition is halted.

[XM_HM_AC_PARTITION_SHUTDOWN] The shutdown event is sent to the offending partition.

[XM_HM_AC_SYSTEM_COLD_RESET] The offending processor is cold reset.

[XM_HM_AC_SYSTEM_WARM_RESET] The offending processor is warm reset.

[XM_HM_AC_SYSTEM_HALT] The offending processor is halted.

[XM_HM_AC_PROPAGATE] The original trap is delivered to the faulting partition. This action can only be used in the partition's health monitoring tables.

Health monitoring logs:

The HM log entries are a "C" structure with the following fields:

```
typedef struct {
    xm_u32_t eventId:13, system:1, reserved:2, moduleId:8,
        partitionId:8;
    xm_u32_t word[5]; /* Payload */
    xmTime_t timeStamp;
} xmHmLog_t;
```

Listing 1.9: core/include/objects/hm.h

See also:

XM_hm_open [Page: 19], XM_hm_read [Page: 20], XM_hm_seek [Page: 21], XM_hm_status [Page: 23]

Chapter 2

Hypercalls

2.1 XM_create_queuing_port

Synopsis: Create a queuing port.

Category:

Standard service.

Declaration:

```

xm_s32_t XM_create_queuing_port (char      *portName,
                                   xm_u32_t  maxNoMsgs,
                                   xm_u32_t  maxMsgSize,
                                   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 can be created by the partition. Note that all the parameters must match the information contained in the configuration file. New ports can not be created dynamically.

The parameters `maxNoMsgs` (the maximum number of messages stored in the channel) and `maxMsgSize` (the largest message) shall match the configuration values of the `XM.CF` file.

It is not an error to create an already created port. The same port descriptor is returned.

Return value:

Upon successful completion, `XM_create_queuing_port()` returns a non-negative integer representing the port descriptor. Otherwise, the function returns a negative value:

[`XM_INVALID_CONFIG`]

- The maximum number of queuing ports has been created.
- No queuing port of the partition is named `portName` in the configuration file.
- `maxMsgSize` is out of range or not compatible with the configuration.
- `direction` is invalid or not compatible with the configuration.

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(0,-1);
}

```

See also:

`XM_send_queuing_message` [Page: 39], `XM_receive_queuing_message` [Page: 33],
`XM_get_queuing_port_status` [Page: 14].

2.2 XM_create_sampling_port

Synopsis: Create a sampling port.

Category:

Standard service.

Declaration:

```

xm_s32_t XM_create_sampling_port (char    *portName,
                                   xm_u32_t    maxMsgSize,
                                   xm_u32_t    direction);

```

Description:

The `XM_create_sampling_port` service is used to create a sampling port. Only those sampling ports specified in the configuration file can be created by a partition. Note that all the parameters must match the information contained in the configuration file. New ports can not be created dynamically.

The parameter `maxMsgSize` (the largest message) shall match the configuration values of the `XM_CF` file.

It is not an error to create an already created port. The same port descriptor is returned.

Return value:

Upon successful completion, `XM_create_sampling_port()` returns a non-negative integer representing the port descriptor. Otherwise, the function returns a negative value:

[`XM_INVALID_CONFIG`]

- The maximum number of sampling ports has been created.
- No sampling port of the partition is named `portName` in the configuration file.
- `maxMsgSize` is out of range or not compatible with the configuration.
- `direction` is invalid or not compatible with the configuration.
- If the port is a system port, the `max_msg_size` parameter does not match the size of the messages defined for those ports.
- The maximum number of open ports exceeded. This error condition should be also raised when compiling the configuration file (`XM_CF`).

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: 64], `XM_read_sampling_message` [Page: 31],

`XM_get_sampling_port_status` [Page: 15].

2.3 XM_disable_irqs

Synopsis: Disable interrupts.

Category:

Standard service. Fast hypercall.

Declaration:

```
void XM_disable_irqs(void);
```

Description:

All native external interrupts and virtual device events are “disabled” for the partition. That is, interrupts are not delivered to the partition. Note that the processor native interrupts are always enabled while a partition is being executed.

Note that the traps caused by the processor and the events generated by an error condition will be delivered to the offending partition asynchronously.

Return value:

This hypercall always succeed.

See also:

[XM_mask_irq](#) [Page: 26], [XM_unmask_irq](#) [Page: 61],

[XM_enable_irqs](#) [Page: 13]

2.4 XM_enable_irqs

Synopsis: Enable interrupts.

Category:

Standard service. Fast hypercall.

Declaration:

```
void XM_enable_irqs(void);
```

Description:

All native external interrupts and virtual device events are “enabled”. Note that this function operates on the virtual state on the partition and not on the native hardware.

If an non-masked interrupt occurs while interrupts, then the highest priority interrupt will then be received by the partition when interrupts are enabled.

Return value:

This hypercall always succeed.

See also:

[XM_mask_irq \[Page: 26\]](#), [XM_unmask_irq \[Page: 61\]](#),

[XM_disable_irqs \[Page: 12\]](#)

2.5 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 (xm_u32_t portDesc,  
                                     xmQueuingPortStatus_t *status  
                                     );
```

Description:

This function fills the structure pointed to by the status parameter, with the following information:

```
typedef struct {  
    xmTime_t validPeriod; // Refresh period.  
    xm_u32_t maxMsgSize; // Max message size.  
    xm_u32_t maxNoMsgs; // Max number of messages.  
    xm_u32_t noMsgs; // Current number of messages.  
    xm_u32_t flags;  
} xmQueuingPortStatus_t;
```

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

Return value:

The return values are:

[XM_OK]

The operation succeeds.

[XM_INVALID_PARAM]

portDesc does not identify an existing destination queuing port, or status is not a valid partition address.

See also:

XM_send_queuing_message [Page: 39], XM_receive_queuing_message [Page: 33],

XM_create_queuing_port [Page: 10].

2.6 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 (xm_u32_t portDesc,  
                                     xmSamplingPortStatus_t *status)  
                                     ;
```

Description:

This function fills the structure pointed to by the status parameter, with the following information:

```
typedef struct {  
    xmTime_t validPeriod; // Refresh period.  
    xm_u32_t maxMsgSize; // Max message size.  
    xm_u32_t flags;  
} xmSamplingPortStatus_t;
```

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

Return value:

[XM_OK]

The operation succeeds.

[XM_INVALID_PARAM]

portDesc does not identify an existing destination queuing port, or status is not a valid partition address.

See also:

[XM_write_sampling_message](#) [Page: 64], [XM_read_sampling_message](#) [Page: 31],

[XM_create_sampling_port](#) [Page: 11].

2.7 XM_get_time

Synopsis: Retrieve the time of the specified clock.

Category:

Standard service.

Declaration:

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

Description:

This function retrieves the number of **microseconds** elapsed since the last hardware reset. The time is stored in the memory pointed to by the pointer *time*.

XtratuM provides the next clocks:

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

Listing 2.3: 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]

The operation succeeds.

[XM_INVALID_PARAM]

If the *clockId* is a non valid virtual clock.

Rationale:

Since the resolution of the clock is 1 microsecond it may happen that the same time value is 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 with 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: 40]

2.8 XM_halt_partition

Synopsis: Terminates a partition.

Category:

Standard service/Supervisor reserved.

Declaration:

```
xm_s32_t XM_halt_partition (xmId_t partitionId);
```

Description:

The `XM_halt_partition()` hypercall causes the termination of the `partitionId` partition. The partition is set in 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 supervisor partitions can halt other partitions than itself. 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 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, and the hypercall does not return. Otherwise, the return value is:

[XM_OK]

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

[XM_INVALID_PARAM]

`partitionId` is not a valid partition identifier.

[XM_PERM_ERROR]

The calling partition is not supervisor and the target partition is not itself.

Usage examples:

```
...
XM_halt_partition(XM_PARTITION_SELF);
/* This code is never executed */
```

See also:

`XM_reset_partition` [Page: 35], `XM_suspend_partition` [Page: 53],
`XM_resume_partition` [Page: 38].

2.9 XM_halt_system

Synopsis: Stop the system.

Category:

Supervisor reserved.

Declaration:

```
xm_s32_t XM_halt_system();
```

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. If the watchdog is armed, when the watchdog timer will expire the board will restart.

This is a supervisor reserved function.

Return value:

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

[XM_PERM_ERROR]

The calling partition is not a supervisor partition.

Rationale:

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

See also:

[XM_reset_system](#) [Page: 37]

2.10 XM_hm_open

Synopsis: Open the health monitoring log stream.

Category:

Standard service.


Declaration:

```
xm_s32_t XM_hm_open ();
```

Description:

This function sets up the internal data structures to allow the calling partition to read from the read health monitoring (hm) log stream.

The contents and the read position of the health monitoring (hm) log stream is not changed.

Multiple calls to this service returns the same stream descriptor, and not a duplicate of it. There is one single XtratuM wide read position attribute, which is shared between all partitions. 

The health monitoring log stream is a single system resource that can be opened by any supervisor partition (this resource is not pre-allocated in the XML configuration file). It is advisable to ensure that only one partition reads from the hm log stream.

Return value:

[XM_OK]

The operation succeeded.

[XM_PERM_ERROR]

The calling partition is not supervisor.

History:

Introduced in XtratuM 2.2.0.

See also:

[XM_hm_read \[Page: 20\]](#), [XM_hm_seek \[Page: 21\]](#), [XM_hm_status \[Page: 23\]](#)

2.11 XM_hm_read

Synopsis: Read a health monitoring log entry.

Category:

Standard service.

Declaration:

```
xm_s32_t XM_hm_read (xmHmLog_t *hmLogPtr);
```

Description:

Read one health monitoring log entry from the current read position into the structure pointed by `hmLogPtr`.

If the read pointer is not at the end of the stream, then it is advanced to the next log entry.

Note that this operation is not destructive. That is, the log entries are not removed from the internal buffer once read. It is possible to retrieve already read entries moving the read position with the hypercall `XM_hm_seek()`.

A health monitoring log entry is a data `xmTraceEvent_t` structure (see [Health Monitoring overview \[Page: 5\]](#)).

Return value:

[XM_OK]

The operation succeeded. The current read position is increased by one.

[XM_PERM_ERROR]

The calling partition is not supervisor.

[XM_INVALID_PARAM]

The `hmLogPtr` address is not a valid partition address.

History:

Introduced in XtratuM 2.2.0.

Usage examples:

```
xmHmLog_t hmLogEntry;
...
if (XM_hm_open() != XM_OK) {
    XM_halt_partition(XM_PARTITION_SELF);
}
while (1) {
    XM_idle_self();
    XM_hm_read(&hmLogEntry);
    ProcessHmEntry(&hmLogEntry);
}
```

See also:

[XM_hm_open \[Page: 19\]](#), [XM_hm_seek \[Page: 21\]](#), [XM_hm_status \[Page: 23\]](#)

2.12 XM_hm_seek

Synopsis: Sets the read position in the health monitoring stream.

Category:

Standard service.

Declaration:

```
xm_s32_t XM_hm_seek (xm_s32_t offset, xm_u32_t whence);
```

Description:

This hypercall repositions the read position of the health monitoring stream to the argument `offset` according to the directive `whence` as follows:

[XM_OBJ_SEEK_SET]

The new read position is `offset` entries from the oldest one stored health monitoring log entry. Only positive values (or zero) of `offset` are valid.

[XM_OBJ_SEEK_CUR]

The new read position is set to the current read position plus the `offset` value. Negative values of `offset` refer to older (previous) log entries. Positive values refer to newer (posterior) log entries. If `offset` is zero then the current position is not changed.


[XM_OBJ_SEEK_END]

The new read position is set to the end of the stream. Negative values of `offset` refer to older (previous) entries. If `offset` is zero then the next read operation will not return a log entry (unless a new hm event is recorded after the completion of this call).

Note that the `offset` value represents the number of log entries, and not the size in bytes.

`offset` shall not be greater than the size of the health monitoring log buffer.

If the computed new read position is beyond the beginning or the end of the stream, then it will be trunked to the start or the end of the stream respectively.

Since the stream is a circular buffer, it may happen that right after the `XM_hm_seek` operation, one or more new hm log entries are stored in the stream. In this case, the pointer may not be positioned at the start or the end of the stream, but a few entries after or before it. To avoid this concurrency issues, it is advisable not to move the pointer too close to the start of the buffer when the buffer is full. 

Return value:

[XM_OK]

The position has been set.

[XM_PERM_ERROR]

The calling partition is not supervisor.

[XM_INVALID_PARAM]

The `offset` or `whence` are not valid.

History:

Introduced in XtratuM 2.2.0.

Usage examples:

```
xmHmLog_t hmBuffer[10];
XM_hm_seek (0, XM_OBJ_SEEK_CUR);
for (x=0; x<10 ;x++){
    /* Read the oldest 10 entries */
    XM_hm_read (&hmBuffer[x]);
}
```

See also:

[XM_hm_open \[Page: 19\]](#), [XM_hm_read \[Page: 20\]](#), [XM_hm_status \[Page: 23\]](#)

2.13 XM_hm_status

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

Category:

Standard service.

Declaration:

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

Description:

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

This service return a stat structure, which contains the following fields:

```
typedef struct {  
    xm_s32_t noEvents;  
    xm_s32_t maxEvents;  
    xm_s32_t currentEvent;  
} xmHmStatus_t;
```

Listing 2.4: core/include/objects/hm.h

The health monitoring log stream shall be open (see XM_hm_open()) before calling this service.

Return value:

[XM_OK]

The operation succeeded.

[XM_PERM_ERROR]

The calling partition is not supervisor.

[XM_INVALID_PARAM]

The hmStatusPtr address is not a valid partition address.

History:

Introduced in XtratuM 2.2.0.

See also:

XM_hm_open [Page: 19], XM_hm_read [Page: 20], XM_hm_seek [Page: 21]

2.14 XM_idle_self

Synopsis: Idles the execution of the calling partition.

Category:

Standard service.

Declaration:

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

Description:

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

Return value: This function always succeeds. No value is returned.

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:

The next example shows how the information reported in the `partitionControlTable_t` data structure can be used to synchronise the execution of a partition with the system cyclic plan.

```
#include <guest.h>

#define HW_IRQS 16

partitionControlTable_t pct;

...

/* 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();
/* Wait the start of the next MAF */
if (pct.tag == 0) { /* If we are in the first slot of
                    the current MAF, then skip it. */
    XM_idle_self();
}
while (pct.tag != 0) {
    XM_idle_self(); /* Skip all the non-starting slots */
}
```

```
}
/* 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.

2.15 XM_mask_irq

Synopsis: Mask an interrupt.

Category:

Standard service. Fast hypercall.

Declaration:

```
xm_s32_t XM_mask_irq (xm_u32_t irq);
```

Description:

This function masks a hardware or extended interrupt.

Only hardware interrupts that had been allocated to the partition (in the configuration file) will be delivered (received) by the partition.

The first 32 interrupt numbers are mapped to the hardware interrupts, and interrupt numbers in the range [32..63] are extended XtratuM interrupts.

Interrupts are mapped in the trap table as follows:

[1 .. 15]

Hardware interrupts. These interrupts trigger the traps from 0x11 to 0x1F.

[31 .. 63]

Extended interrupts. These interrupts trigger the traps from 0x100 to 0x11f.

All extended interrupts can be masked and unmasked.

Return value:

[XM_OK]

The interrupt line has been masked.

[XM_INVALID_PARAM]

The `irq` is not owned by the calling partition or it is an invalid interrupt number.

See also:

`XM_unmask_irq` [Page: 61]

2.16 XM_memory_copy

Synopsis: Copy copies data from/to address spaces.

Category:

Standard service/Supervisor reserved.

Declaration:

```

xm_s32_t XM_memory_copy (xmId_t destId, xm_u32_t destAddr,
                        xmId_t srcId, xm_u32_t srcAddr,
                        xm_u32_t size);


```

Description:

This function copies data from/to address spaces. The addresses (source and/or destination) can point to both memory or 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 `destAddr` in the memory space of `destId` 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, care must be taken to avoid breaking temporal isolation.
- This function has been implemented using one object descriptor (associated with the `/dev/mem` object). The maximum number of object descriptors per partition is a source code configuration parameter, and shall be large enough. 
- If the source or destination addresses do not belong to the space of a partition (for example, ROM areas) then the `XM_SYSTEM_ID` shall be used.
Only supervisor partitions are allowed to perform a copy for/to other address space than its own (i.e. other partitions or system space).
- 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]

The operation succeeds.

[XM_INVALID_PARAMS]

- `destId` or `srcId` are not valid partition Id's,
- `destAddr` does not belong to the `destId` partition,
- `srcAddr` does not belong to the `srcId` partition.

[XM_INVALID_CONFIG]

The maximum number of open object descriptors has been exceeded. If this happens, then XtratuM shall be reconfigured to increase this limit and recompiled accordingly.

[XM_PERM_ERROR]

The calling partition is not supervisor.

Rationale:

Since the LEON2 does not have MMU, both address (`destAddr` and `srcAddr`) are physical memory addresses. Partition identifiers are not needed, but has been included for future compatibility.

Partition identifiers are not checked in the current implementation.

2.17 XM_multicall

Synopsis: Execute a sequence of hypercalls.

Category:

Standard service. Optimization. [Not ported to LEON2]

Declaration:

```
xm_s32_t XM_multicall(void *buffer, xm_s32_t nr);
```

Description:

In some cases, the partition has to perform a large sequence of hypercalls to perform an operation. For example, several `XM_sparcv8_outport()` calls may be needed to program a single peripheral operation; another common case is when managing memory maps (not implemented in this version).

In order to reduce the overhead caused by the hypercall mechanism, XtratuM provides this hypercall to **pack** several hypercalls in a buffer, and then execute them using a single hypercall.

This hypercall does not provide new functionality, it is only a mechanism that can be used to improve the performance.

Return value: TBD.

2.18 XM_partition_get_status

Synopsis: Get the current status of a partition.

Category:

Standard service.

Declaration:

```
xm_s32_t XM_partition_get_status (xmId_t      id,
                                xmPartitionStatus_t *state);
```

Description:

Returns the current state of the partition referenced by id into the state structure.

The xmPartitionStatus_t is a data type contains the following fields:

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

Listing 2.5: core/include/objects/status.h

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]

The operation succeeded.

[XM_PERM_ERROR]

The calling partition is not supervisor.

[XM_INVALID_PARAM]

id is not a valid identifier, or state is not a valid partition address.

History:

Introduced in XtratuM 2.2.0.

See also:

XM_system_get_status [Page: 54].

2.19 XM_read_sampling_message

Synopsis: Reads a message from the specified sampling port.

Category:

Standard service.

Declaration:

```
xm_s32_t XM_read_sampling_message (xm_s32_t portDesc,
                                   void      *msgPtr,
                                   xm_u32_t size,
                                   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 `size` 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:

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

[`XM_INVALID_PARAM`]

- `portDesc` does not identify a valid port.
- The specified sampling port is not configured as `XM_PORT_DESTINATION`.
- The value of `size` is zero.

[`XM_INVALID_CONFIG`]

The `size` is not compatible with the configuration of the specified port.

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",
                                   30,
                                   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 \[Page: 11\]](#), [XM_write_sampling_message \[Page: 64\]](#),
[XM_get_sampling_port_status \[Page: 15\]](#).

2.20 XM_receive_queuing_message

Synopsis: Receive a message from the specified queuing port.

Category:

Standard service.

Declaration:

```
xm_s32_t XM_receive_queuing_message (xm_s32_t portDesc,  
                                     void      *msgPtr,  
                                     xm_u32_t  size,  
                                     xm_u32_t  *flags);
```

Description:

If the port channel is not empty, then the oldest message is retrieved from the channel. At most `size` bytes are copied into the buffer pointed by `msgPtr`. The actual number of bytes received is returned by the call (if the hypercall succeeds).

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 (the parameter `size` is smaller than the actual size of the message in the channel).

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:

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

[`XM_NOT_AVAILABLE`]

The channel is empty or the port is not connected to a channel in the `XM.CF` configuration file.

[`XM_INVALID_PARAM`]

`portDesc` does not identify an existing queuing port or it was not configured as a destination port; the value of `size` is zero.

[`XM_INVALID_CONFIG`]

The `size` is not compatible with the configuration of the specified port.

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.

See also:

`XM_create_queuing_port` [Page: 10], `XM_send_queuing_message` [Page: 39],

`XM_get_queuing_port_status` [Page: 14].

2.21 XM_request_irq

Synopsis: Request to receive an interrupt.

Category:

Removed.

Declaration:

```
xm_s32_t XM_request_irq (xm_u32_t irq)
```

Description:

This hypercall is no longer available.

Rationale:

This service was initially provided for compatibility with the ARINC-653 standard. It checks that the requested interrupt line was effectively allocated to the calling partition.

Since all hypercalls that deal with interrupts also perform that check, this service is redundant.

2.22 XM_reset_partition

Synopsis: Reset a partition.

Category:

Standard service/Supervisor reserved.

Declaration:

```

xm_s32_t XM_reset_partition (xm_s32_t partitionId,
                             xm_u32_t resetMode,
                             xm_u32_t status);

```

Description:

The partition `partitionId` is reset. There are two ways to reset a partition depending on the `resetMode` value:


XM_WARM_RESET:

1. The `resetCounter` field of the partition information table (PIT) is incremented.
2. The parameter `status` is copied in the field `resetStatus` of the partition information table
3. The program counter is set to the partition entry point.
4. The partition is set in normal/ready state.

XM_COLD_RESET:

1. All communication ports are closed.
2. The PCT and PIT data structures are initialised in the partition memory space.
3. The `resetCounter` field of the partition information table (PIT) is set to zero.
4. The parameter `status` is copied in the field `resetStatus` of the partition information table
5. The program counter is set to the partition entry point.
6. The partition is set in 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 the PIT and PCT) than before the reset. 

Only supervisor partitions can reset other partitions than itself. 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 his 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]

A non supervisor partition attempted to reset other partition.

Usage examples:

```
...  
XM_reset_partition(0, -9);  
/* This code is never reached */
```

See also:

[XM_halt_partition \[Page: 17\]](#), [XM_suspend_partition \[Page: 53\]](#),
[XM_resume_partition \[Page: 38\]](#), [XM_memory_copy \[Page: 27\]](#)

2.23 XM_reset_system

Synopsis: Reset the system.

Category:

Supervisor reserved.

Declaration:

```
xm_s32_t XM_reset_system(xm_u32_t mode);
```

Description:


The system is reset immediately. There are two ways to reset the system depending on the mode value:

XM_WARM_RESET:

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

XM_COLD_RESET:

XtratuM unconditionally jumps back to the resident software entry point. This value shall be specified attribute `/SystemDescription/ResidentSw/@entryPoint` of the XML configuration file. If the `./ResidentSw/@entryPoint` attribute is not specified then XtratuM will jump to address `0x000000`.

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 supervisor partition.

[XM_INVALID_PARAM]

mode is not a valid mode.

Rationale:

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

See also:

`XM_halt_system` [Page: 18]

2.24 XM_resume_partition

Synopsis: Resume the execution of a partition.

Category:

Supervisor reserved.

Declaration:

```
xm_s32_t XM_resume_partition (xmId_t partitionId);
```

Description:

Resumes the execution of the target partition, `partitionId`. 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 supervisor 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 does not have supervisor rights.

See also:

`XM_suspend_partition` [Page: 53].

2.25 XM_send_queuing_message

Synopsis: Send a message in the specified queuing port.

Category:

Standard service.

Declaration:

```
xm_s32_t XM_send_queuing_message (xm_s32_t portDesc,  
                                  void      *msgPtr,  
                                  xm_u32_t  size);
```

Description:

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

Return value:

[XM_OK]

The message has been successfully written into the port.

[XM_NOT_AVAILABLE]

Insufficient space in the channel or the port is not connected to a channel in the XM_CF configuration file.

[XM_INVALID_PARAM]

portDesc does not identify an existing destination queuing port; or size is zero.

[XM_INVALID_CONFIG]

The size is not compatible with the configuration of the specified port.

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.

See also:

[XM_create_queuing_port](#) [Page: 10], [XM_receive_queuing_message](#) [Page: 33],

[XM_get_queuing_port_status](#) [Page: 14].

2.26 XM_set_timer

Synopsis: Arm a timer.

Category:

Standard service.

Declaration:

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

Description:

If `interval` is zero, then the timer is armed only once. The timer associated with the virtual clock `clockId` is armed 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 repeats until the timer is re-armed.

If the specified `absTime` time has already passed, the function succeeds and the 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 was pending timer interrupts, then the interrupt 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]

The timer has been successfully armed.

[XM_INVALID_PARAM]

- The specified `clockId` is a non valid virtual clock (not a valid clock or the clock can not be used to arm timers).

Rationale:

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

On systems where the cost of reprogramming the timer hardware is low (the case of the LEON2 board), the one shot timer mode is the best technique because it provides an high resolution and small overhead.

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_unmask_irq(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_unmask_irq(XM_VT_EXT_HW_TIMER);
```

See also: [XM_get_time](#) [Page: 16]

2.27 XM_shutdown_partition

Synopsis: Send a shutdown interrupt to a partition.

Category:

Standard service/Supervisor reserved.

Declaration:

```
xm_s32_t XM_shutdown_partition (xmId_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, the target partition shall close and terminate the ongoing tasks and finally call the `XM_halt_partition` [Page: 17] hypercall. XtratuM does not control the state the target partition after a shutdown request.

Only supervisor partitions can invoke this service to shutdown other partitions than itself. 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 supervisor and the target partition is not itself.

See also:

`XM_reset_partition` [Page: 35], `XM_suspend_partition` [Page: 53],

`XM_resume_partition` [Page: 38].

2.28 XM_sparcv8_atomic_add

Synopsis: Atomic add.

Category:

Standard service. Fast hypercall.

Declaration:

```
void XM_sparcv8_atomic_add(xm_s32_t *target, xm_s32_t delta);
```

Description:

Adds the value `delta` to the value stored in `target` in an atomic manner.

Arithmetic overflow or underflow of the target value is not checked.

Return value:

This hypercall always succeeds.

If an invalid target pointer is given then a trap is raised (TBD).

Rationale:

SparcV8 processor lacks support for complex atomic operations. Since a partition is always executed with interrupts enabled, atomic operations have to be provided by XtratuM.

See also:

`XM_sparcv8_atomic_and` [Page: 44], `XM_sparcv8_atomic_or` [Page: 45]

2.29 XM_sparcv8_atomic_and

Synopsis: Atomic bitwise AND.

Category:

Standard service. Fast hypercall.

Declaration:

```
void XM_sparcv8_atomic_and(xm_u32_t *target, xm_u32_t bits);
```

Description:

Performs the bitwise AND of bits to the value stored in target in an atomic manner.

Return value:

This hypercall always succeeds.

If an invalid target pointer is given then a trap is raised (TBD).

Rationale:

This service can be used to implement the atomic clear bit operation:

```
static inline void XM_sparcv8_atomic_clearbit(xm_u32_t *target,
                                              xm_u32_t pos){
    XM_sparcv8_atomic_and(target, ~(1 << pos));
}
```

See also:

[XM_sparcv8_atomic_add \[Page: 43\]](#), [XM_sparcv8_atomic_or \[Page: 45\]](#)

2.30 XM_sparcv8_atomic_or

Synopsis: Atomic bitwise OR.

Category:

Standard service. Fast hypercall.

Declaration:

```
void XM_sparcv8_atomic_or(xm_u32_t *target, xm_u32_t bits);
```

Description:

Performs the bitwise OR of bits to the value stored in target in an atomic manner.

Return value:

This hypercall always succeeds.

If an invalid target pointer is given then a trap is raised (TBD).

Rationale:

This service can be used to implement the atomic set bit operation:

```
static inline void XM_sparcv8_atomic_setbit(xm_u32_t *target,
                                             xm_u32_t pos){
    XM_sparcv8_atomic_or(target, (1 << pos));
}
```

See also:

[XM_sparcv8_atomic_add \[Page: 43\]](#), [XM_sparcv8_atomic_and \[Page: 44\]](#)

2.31 XM_sparcv8_flush_regwin

Synopsis: Save the contents of the register window.

Category:

Standard service. Assembly hypercall.

Declaration:

```
void XM_sparcv8_flush_regwin(void);
```

Description:

Saves the contents of all the register windows but the active one when the hypercall is invoked.

This hypercall assumes that the stack registers contain valid (and correct) addresses.

Return value:

No value is returned.

Rationale:

This function is needed if the partition code has to implement threads.

A counterpart function to reload the register window is not needed because XtratuM manages transparently the stack. That is, it will be reloaded automatically when the register window underflows.

2.32 XM_sparcv8_get_flags

Synopsis: Get the ICC flags from the PSR processor register.

Category:

Standard service. Assembly hypercall.

Declaration:

```
xm_u32_t XM_sparcv8_get_flags(void);
```

Description:

Returns the four ICC (Integer Condition Codes) fields of the PSR (Processor Status Register). The bits 20 to 23 contains the ICC flags.

2.33 XM_sparcv8_inport

Synopsis: Read from a hardware I/O port.

Category:

Standard service. Fast hypercall.

Declaration:

```
xm_s32_t XM_sparcv8_inport (xm_u32_t portAddr,  
                           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 SPARC v8 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_sparcv8_outport` [Page: 50], `XM_memory_copy` [Page: 27]

2.34 XM_sparcv8_iret

Synopsis: Return from an interrupt.

Category:


Standard service. Assembly hypercall.

Declaration:

```
XM_sparcv8_iret();
```

Description:

Emulates a `rett` (RETurn from Trap) assembly instruction.

This service should be called at the end of an exception handler; otherwise the result is undetermined. The program counter of the partition will be corrupted. 

The ICC fields of the PSR are restored with the content of the register 10. 

The hypercall is provided as an assembly macro.

Return value:

The partition will continue with the normal execution flow, previous to the last attended trap.

Rationale:

When a trap is triggered, the current register window is decremented by one; and the processor register contain the following values:

- The register 10 contains a copy of the ICC flags.
- The register 11 contains the PC (program counter).
- The register 12 contains the nPC (next program counter).

Usage examples:

```
exception_handler_asm:
    call exception_handler
    nop
    XM_sparcv8_iret()

/* Code expanded in the trap table. */
#define BUILD_TRAP(trapnr) \
    sethi %hi(exception_handler_asm), %l4 ;\
    jmpl %l4 + %lo(exception_handler_asm), %g0 ;\
    mov trapnr, %o0 ;\
    nop
```

2.35 XM_sparcv8_outport

Synopsis: Write in a hardware I/O port.

Category:

Standard service. Fast hypercall.

Declaration:

```
xm_s32_t XM_sparcv8_outport (xm_u32_t portAddr,
                             xm_u32_t value);
```

Description:

This service provides direct hardware access to the peripherals. On response to this call, XtratuM performs the write of `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="0xc0000008" noPorts="2"/>
    </IoPorts>
  </HwResources>
</Partition>
```

In this example, the ports `0xc0000008` and `0xc0000009` 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. In order to implement this feature, XtratuM executed the code below:

```
oldValue=LoadIoReg(port);
StoreIoReg(port, ((oldValue&~(mask))|(value&mask)));
```

Listing 2.6: `core/kernel/sparcv8/hypercalls.c`



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 SPARC v8 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_sparcv8_inport [Page: 48], XM_memory_copy [Page: 27]

2.36 XM_sparcv8_set_flags

Synopsis: Set the ICC flags on the PSR processor register.

Category:

Standard service. Assembly hypercall.

Declaration:

```
void XM_sparcv8_set_flags(xm_u32_t flags);
```

Description:

Sets four ICC (Integer Condition Codes) fields of the PSR (Processor Status Register). The bits 20 to 23 of the flags parameter are copied into the ICC flags of the PSR processor register.

2.37 XM_suspend_partition

Synopsis: Suspend the execution of a partition.

Category:

Standard service/Supervisor reserved.


Declaration:

```
xm_s32_t XM_suspend_partition (xmId_t partitionId);
```

Description:

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

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

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

Only supervisor partitions can suspend other partitions than itself. Any partition can halt 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 suspend state.

[XM_INVALID_PARAM]

`partitionId` is not a valid partition identifier.

[XM_PERM_ERROR]

The calling partition is not supervisor and the target partition is not itself.

See also:

`XM_reset_partition` [Page: 35], `XM_halt_partition` [Page: 17],

`XM_resume_partition` [Page: 38].

2.38 XM_system_get_status

Synopsis: Get the current status of the system.

Category:

Standard service.

Declaration:

```
xm_s32_t XM_system_get_status (xmSystemStatus_t *state);
```

Description:

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

The xmSystemStatus_t is a data type which contains the following fields:

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

Listing 2.7: core/include/objects/status.h

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]

The operation succeeded.

[XM_PERM_ERROR]

The calling partition is not supervisor.

[XM_INVALID_PARAM]

state is not a valid partition address.

History:

Introduced in XtratuM 2.2.0.

See also:

XM_partition_get_status [Page: 30].

2.39 XM_trace_event

Synopsis: Records a trace entry.

Category:

Standard service.

Declaration:

```
xm_s32_t XM_trace_event (xm_u32_t  bitmask,
                        xmTraceEvent_t *event);
```

Description:

This service records trace event pointed by event into the partition's trace stream if the logical and operation between the bitmak parameter and the /XMHypervisor/PartitionTable/Partition/-Trace/@bitmask attribute of the configuration file.

Each partition has its own trace stream to store the trace events generated by the partition. This trace stream buffer has to be configured in the XM.CF configuration file, and is automatically opened after a reset.

A health monitoring log entry is a data xmTraceEvent_t structure.

Where the xmTraceOpCode_t is a 32bits word with the following bit fields:

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

Listing 2.8: core/include/objects/trace.h

partitionId

Constains the identifier of the calling partition. It is filled by XtratuM.

moduleId

User defined field.

criticality

Defines the importance of the trace event.

code

An number which identifies cause of the trace.

The event shall be aligned to 8 bytes.

If the trace stream is stored in non-volatile memory, then the internal trace stream may become full. In this case, the oldest traced event is overwritten.

Return value:

[XM_OK]

The trace event has been stored successfully.

[XM_INVALID_PARAM]

event address is not a valid.

History:

Introduced in XtratuM 2.2.0.

See also:

`XM_trace_open` [Page: 57], `XM_trace_read` [Page: 58], `XM_trace_seek` [Page: 59], `XM_trace_status` [Page: 60].

2.40 XM_trace_open

Synopsis: Open a trace stream.

Category:

Standard service.

Declaration:

```
xm_s32_t XM_trace_open (xmId_t id);
```

Description:

Returns the trace stream descriptor of the partition whose id is: id. If the id parameter is XM_HYPERVISOR_ID, then the trace stream of XtratuM is returned.

The trace events generated by a partition are stored in a trace stream with the same name than the partition that created the traces. The trace events generated by XtratuM are stored in the trace stream named "xm".

The first time the trace stream is opened, the read position is set to the oldest trace event.

Opening multiple times the same trace stream returns the same stream descriptor, and not a duplicate of it. Therefore, the read position is not changed as a result of an open operation.



The trace stream has to be allocated (in the XML configuration file) to the partition that opens it. Otherwise an error is returned. At most one partition can open a trace stream.

Return value:

Return a positive number which represents the trace stream descriptor. A negative number is returned in case of an error. The returned error values are:

[XM_PERM_ERROR]

The calling partition is not supervisor.

[XM_INVALID_PARAM]

The id is not a valid partition address, or a valid trace stream name.

History:

Introduced in XtratuM 2.2.0.

See also:

[XM_trace_event](#) [Page: 55], [XM_trace_read](#) [Page: 58], [XM_trace_seek](#) [Page: 59], [XM_trace_status](#) [Page: 60].

2.41 XM_trace_read

Synopsis: Read a trace event.

Category:

Standard service.

Declaration:

```
xm_s32_t XM_trace_read (xm_s32_t  traceStream,  
                        xmTraceEvent_t *traceEventPtr);
```

Description:

Attempts to read one trace event from the stream traceStream.

If there was unread events in the stream, a positive value is returned **and the read position is advanced by one**. If there was no new trace events to be read, then a zero value is returned.

Note that this operation is not destructive. That is, the trace events are not removed from the internal stream once read. It is possible to retrieve old events with the hypercall XM_trace_seek().

A health monitoring log entry is a data xmTraceEvent_t structure.

Return value:

[XM_OK]

The operation succeeds.

[XM_PERM_ERROR]

The calling partition is not supervisor.

[XM_INVALID_PARAM]

The traceEventPtr address is not a valid partition address; or traceStream is not a valid trace stream descriptor.

History:

Introduced in XtratuM 2.2.0.

See also:

XM_trace_event [Page: 55], XM_trace_open [Page: 57], XM_trace_seek [Page: 59], XM_trace_status [Page: 60].

2.42 XM_trace_seek

Synopsis: Sets the read position in a trace stream.

Category:

Standard service.

Declaration:

```
xm_s32_t XM_trace_seek (xm_s32_t traceStream,
                       xm_s32_t offset,
                       xm_u32_t whence);
```

Description:

This hypercall repositions the read position of the open file associated with the trace stream descriptor `traceStream` to the argument `offset` according to the directive `whence` as follows:

[XM_OBJ_SEEK_SET]

The new read position is `offset` events from the oldest one stored in the given stream. Only positive values (or zero) of `offset` are valid.

[XM_OBJ_SEEK_CUR]

The new read position is set to the current read position plus the `offset` value. Negative values of `offset` refer to older (previous) trace events. Positive values refer to newer (posterior) events. If `offset` is zero then the current position is not changed.


[XM_OBJ_SEEK_END]

The new read position is set to the end of the stream. Negative values of `offset` refer to older (previous) events. If `offset` is zero then the next read operation will not return a trace event (unless a new event is recorded after the completion of this call).

Note that the `offset` value represents the number of trace events, and not the size in bytes.

`offset` has to be no greater than the size of the internal buffer stream.

If the computed new read position is beyond the beginning or the end of the stream, then it will be trunked to the start or the end of the stream respectively.

Since the stream is a circular buffer, it may happen that right after the `XM_trace_seek()` operation, one or more new trace events are recorded in the stream. In this case, the pointer may not be positioned at the start or the end of the stream, but a few entries after or before it. To avoid this concurrency issues, it is advisable not to move the pointer too close to the start of the buffer when the buffer is full. 

Return value:

[XM_OK]

The operation succeeds.

[XM_PERM_ERROR]

The calling partition is not supervisor.

[XM_INVALID_PARAM]

The `offset` or `whence` are not valid. Or `traceStream` is not a valid trace stream descriptor.

History:

Introduced in XtratuM 2.2.0.

See also:

`XM_trace_event` [Page: 55], `XM_trace_open` [Page: 57], `XM_trace_read` [Page: 58], `XM_trace_status` [Page: 60].

2.43 XM_trace_status

Synopsis: Get the status of a trace stream.

Category:

Standard service.

Declaration:

```
xm_s32_t XM_trace_status (xm_s32_t      traceStream,  
                          xmTraceStatus_t *traceStatusPtr);
```

Description:

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

This service returns a structure which contains the following fields:

```
typedef struct {  
    xm_s32_t noEvents;  
    xm_s32_t maxEvents;  
    xm_s32_t currentEvent;  
} xmTraceStatus_t;
```

Listing 2.9: core/include/objects/trace.h

Return value:

[XM_OK]

The operation succeeded.

[XM_PERM_ERROR]

The calling partition is not supervisor.

[XM_INVALID_PARAM]

The `traceStatusPtr` address is not a valid partition address; or `traceStream` is not a valid trace stream descriptor.

History:

Introduced in XtratuM 2.2.0.

See also:

`XM_trace_event` [Page: 55], `XM_trace_open` [Page: 57], `XM_trace_read` [Page: 58],

`XM_trace_seek` [Page: 59].

2.44 XM_unmask_irq

Synopsis: Unmask an interrupt.

Category:

Standard service. Fast hypercall.

Declaration:

```
xm_s32_t XM_mask_irq (xm_u32_t irq);
```

Description:

This function unmaskes a hardware or extended interrupt.

Only interrupts that had been allocated to the partition (in the configuration file) will be delivered (received) by the partition.

See the XM_mask_irq [Page: 26] man page for more information.

Return value:

[XM_OK]

The interrupt line has been unmasked.

[XM_INVALID_PARAM]

The irq is not owned by the calling partition or it is an invalid interrupt number.

See also:

XM_mask_irq [Page: 26].

2.45 XM_write_console

Synopsis: Print a string in the hypervisor console.

Category:

Standard service.

Declaration:

```
xm_s32_t XM_write_console (char *str, xm_s32_t length);
```

Description:

Writes the up to `length` bytes from the buffer pointed `str` 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 developing and testing purposes, and should not be used in real operation.

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

Return value:

This function always succeeds, returning the number of characters written.

Usage examples:

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

2.46 XM_write_register32

Synopsis: Modify a processor control register.

Category:


Standard service.

Declaration:

```
xm_s32_t XM_write_register32(xm_s32_t register, xm_u32_t value)
;
```

Description:

This para-virtualises the processor. The parameter `register` identifies the processor control register to be modified with the new value. The result of this service depends on the target register.

A wrong value may have fatal consequences on the calling partition. Since this service operates in the virtual processor, no effects can be produced on other partitions or the hypervisor. 

Below is the list of valid processor registers:

```
#define XMTBR_REG32 0
```

Listing 2.10: core/include/sparcv8/hypercalls.h

Return value:

[XM_OK]

The requested operation has been successful.

[XM_INVALID_PARAM]

- The `register` is not a valid register of this architecture.
- The given `value` is not a valid value for the specified register.

2.47 XM_write_sampling_message

Synopsis: Writes a message in the specified sampling port.

Category:

Standard service.

Declaration:

```
xm_s32_t XM_write_sampling_message (xm_s32_t portDesc,  
                                   void      *msgPtr,  
                                   xm_u32_t  size);
```

Description:

The message is atomically copied into the internal XtratuM buffer of the channel.

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

Return value:

[XM_OK]

If the message has been successfully written into internal buffer.

[XM_INVALID_PARAM]

`portDesc` does not identify an existing sampling port or it is not a destination port; or `size` is zero.

[XM_INVALID_CONFIG]

The size is not compatible with the configuration of the specified port.

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) 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: 11], `XM_read_sampling_message` [Page: 31],

`XM_get_sampling_port_status` [Page: 15].