XtratuM Hypervisor for SPARCv8

Reference Manual

Miguel Masmano, Javier O. Coronel, Alfons Crespo, Patricia Balbastre

April, 2016
Reference: 14-036.008.sum.rm

DOCUMENT CONTROL PAGE

TITLE: XtratuM Hypervisor for SPARCv8: Reference Manual

AUTHOR/S: Miguel Masmano, Javier O. Coronel, Alfons Crespo, Patricia Balbastre

LAST PAGE NUMBER: 104

VERSION OF SOURCE CODE: XtratuM for SPARCv8()

REFERENCE ID: 14-036.008.sum.rm

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

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

REFERENCING THIS DOCUMENT:

```
@techreport {14-036.008.sum.rm,
        title = {XtratuM Hypervisor for SPARCv8:  Reference Manual},
        author = { Miguel Masmano and  Javier O. Coronel and  Alfons Crespo and  Patricia
        Balbastre},
        institution = {Universidad Politécnica de Valencia},
        number = {14-036.008.sum.rm},
        year={April, 2016},
}
```

Changes:

Version	Date	Comments
0.1	February, 2010	[xm-3-reference-023b] Code version 3.1.0.

Version	Date	Comments
0.2	May 2010	[xm-3-reference-023c] Code version 3.1.2 Renamed: • XM_system_get_status() → XM_get_system_status(). • XM_partition_get_status() → XM_get_partition_status(). Changed: • XM_unmask_irq() → XM_clear_irqmask() • XM_mask_irq() → XM_set_irqmask() • XM_enable_irqs() → XM_sparcv8_set_pil() • XM_disable_irqs() → XM_sparcv8_clear_pil() Added: • XM_clear_irqpend() • XM_set_irqpend() • XM_get_partition_mmap() • XM_get_sampling_port_info() • XM_get_queuing_port_info() • XM_sparcv8_set_psr() • XM_sparcv8_get_psr() • XM_sparcv8_flush_tlb() • XM_sparcv8_flush_cache() • XM_get_partition_mmap() • XM_set_plan() • XM_get_plan_status() Removed: • XM_get_physmem_map()
0.3	October, 2010	[xm-3-reference-023d] Code version 3.2.0. Added: • XM_update_page32() • XM_set_page_type()
0.4	March, 2011	[xm-3-reference-023e] Code version 3.2.0. Updates the definition of the hypercalls.
1.0	September, 2011	[xm-3-reference-023f] Code version 3.3.0. Added: • a new hypercall XM_sparcv8_ctrl_winflow() • Final release
1.0	December, 2011	[xm-3-reference-023g] Code version 3.3.2. Added: • introduction reorganisation. • reorganization of the hypercalls by category. New hypercalls added: • XM_set_cache_state() • XM_get_gid_by_name() • XM_set_partition_opmode() • XM_hm_get_app_error() • XM_hm_raise_app_error()

Version	Date	Comments
1.1	March, 2012	[xm-3-reference-023h] Code version 3.3.2b. Updated: New hypercalls considered obsoletes: • XM_invld_tlb() (by error this hypercall was named as XM_sparc_flush_tlb()) • XM_set_page_type() • XM_update_page32() Modification of the return values of the XM_set_cache_state() hypercall.
1.2	July, 2012	[xm-3-reference-] Code version 3.9.0. Updated: New hypercalls added: • XM_hm_raise_event() New hypercalls ported from XtratuM 2: • XM_disable_irqs() • XM_enable_irqs() • XM_sparc_disable_sdp() • XM_sparc_outport_sdp() • XM_synchronize() Hypercalls considered obsoletes and removed: • XM_hm_seek() • XM_trace_open() • XM_trace_seek() • XM_read_console()
1.3	September, 2012	[xm-3-reference-] Code version 3.9.1. • HM log structure restored.
1.4	March, 2013	[xm-3-reference-] Code version 3.9.3. XEF services documented: • XEF_parse_file() • XEF_load_file() • XEF_load_custom_file()
1.5	September, 2013	[fnts-xm-rm-3a] Reference changed using FentISS convention.
1.6	November, 2013	[fnts-xm-rm-3b] Code Version 3.9.6. SMP hypercalls documented: • XM_get_vcpuid() • XM_halt_vcpu() • XM_reset_vcpu() Hypercall XM_shutdown_partition() marked as obsolete.
1.7	December, 2014	[14-036.008.sum.rm] Code Version 1.0.0. Memory Map Table (MMT) structure added to partition management types. Interrupt types added to interrupt management types. Hypercalls updated: • XM_get_sampling_port_info() • XM_get_queuing_port_info() Hypercalls removed (obsolete): • XM_synchronize()
1.8	January, 2015	[14-036.008.sum.rm] Code Version 1.0.2. Fixed: • SPR-080115-01

Version	Date	Comments
1.9	May, 2015	[14-036.008.sum.rm] Fixed: • SPR-280115-01 • SPR-280115-04 • SPR-110215-01 • SPR-110215-02 • SPR-120215-01 • SPR-260215-01 • SPR-270215-07 • SPR-020315-01 • SPR-050315-01
2.0	June, 2015	[14-036.008.sum.rm] Fixed: • SPR-180615-02
2.1	September, 2015	[14-036.008.sum.rm] Fixed: • SPR-280715-01 • SPR-160915-01 • SPR-021015-01
2.1	November, 2015	[14-036.008.sum.rm]. Code version 1.0.6. • [SPR-221015-01] 1.2.1 updated. • [SPR-061115-01] 2.1.3 updated. • [SPR-241115-01] 2.12.12 switched with 2.12.8. • [SPR-241115-02] 2.1.3 updated. • [SPR-021215-01] 3.1.3 updated. • [SPR-111215-03] 2.8.1 updated.
2.2	April, 2016	[14-036.008.sum.rm]. Code version 1.0.7. Updated: • [SPR-080216-01] 2.12.5, 2.12.11 updated. • [SPR-180216-02] 2.7.1 return values updated. Added: • [CP-250216-01] 2.4.2 Added `XM_reload_watchdog()`

Contents

Preface xi

1 Introduction 1

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 . . . 10

1.9.1 Trace types . . . 10
1.9.2 Trace events symbols . . . 11
1.10 Interrupt management types . . . 11
1.10.1 Interrupt management symbols . . . 11
1.10.2 Mapping of interrupts to traps . . . 14

2 Hypercalls 15
2.1 System Management . . . 16
2.1.1 XM_get_system_status . . . 16
2.1.2 XM_halt_system . . . 17
2.1.3 XM_reset_system . . . 18
2.2 Partition Management . . . 19
2.2.1 XM_get_partition_mmap . . . 19
2.2.2 XM_get_partition_status . . . 20
2.2.3 XM_halt_partition . . . 21
2.2.4 XM_idle_self . . . 22
2.2.5 XM_params_get_PCT . . . 24
2.2.6 XM_reset_partition . . . 25
2.2.7 XM_resume_partition . . . 27
2.2.8 XM_shutdown_partition . . . 28
2.2.9 XM_suspend_partition . . . 29
2.3 Multicore Management . . . 30
2.3.1 XM_get_vcpuid . . . 30
2.3.2 XM_halt_vcpu . . . 31
2.3.3 XM_reset_vcpu . . . 32
2.4 Time Management . . . 33
2.4.1 XM_get_time . . . 33
2.4.2 XM_reload_watchdog . . . 35
2.4.3 XM_set_timer . . . 36
2.5 Plan Management . . . 38
2.5.1 XM_get_plan_status . . . 38
2.5.2 XM_switch_sched_plan . . . 39
2.6 Inter-Partition Communication . . . 40
2.6.1 XM_create_queuing_port . . . 40
2.6.2 XM_create_sampling_port . . . 42
2.6.3 XM_get_commport_bitmap . . . 43
2.6.4 XM_get_queuing_port_info . . . 44

2.6.5 XM_get_queuing_port_status 45
2.6.6 XM_get_sampling_port_info 46
2.6.7 XM_get_sampling_port_status 47
2.6.8 XM_read_sampling_message 48
2.6.9 XM_receive_queuing_message 50
2.6.10 XM_send_queuing_message 52
2.6.11 XM_write_sampling_message 54
2.7 Memory Management 56
2.7.1 XM_memory_copy 56
2.7.2 XM_set_cache_state 58
2.8 Health Monitor Management 59
2.8.1 XM_hm_raise_event 59
2.8.2 XM_hm_read 60
2.8.3 XM_hm_status 61
2.9 Trace Management 62
2.9.1 XM_trace_event 62
2.9.2 XM_trace_read 63
2.9.3 XM_trace_status 64
2.10 Interrupt Management 65
2.10.1 XM_clear_irqmask 65
2.10.2 XM_clear_irqpend 66
2.10.3 XM_disable_irqs 67
2.10.4 XM_enable_irqs 68
2.10.5 XM_raise_ipvi 69
2.10.6 XM_route_irq 70
2.10.7 XM_set_irqmask 71
2.10.8 XM_set_irqpend 72
2.11 Miscelaneous 73
2.11.1 XM_get_gid_by_name 73
2.11.2 XM_write_console 74
2.12 Sparcv8 specific 75
2.12.1 XM_sparc_clear_pil 75
2.12.2 XM_sparc_ctrl_winflow 76
2.12.3 XM_sparc_disable_sdp 77
2.12.4 XM_sparc_flush_regwin 78
2.12.5 XM_sparc_get_psr 79
2.12.6 XM_sparc_inport 80

2.12.7 XM_sparc_outport 82
2.12.8 XM_sparc_outport_msg 84
2.12.9 XM_sparc_outport_sdp 86
2.12.10 XM_sparc_set_pil 87
2.12.11 XM_sparc_set_psr 88
2.12.12 sparc_iret_nr 89
2.12.13 sparc_write_tbr_nr 91
2.13 Obsolete 92
2.13.1 XM_sparc_flush_cache 92
2.14 Library functions 93
2.14.1 XEF_load_custom_file 93
2.14.2 XEF_load_file 94
2.14.3 XEF_parse_file 95
2.14.4 init_libxm 97

3 Library functions **99**
3.1 XEF functions 100
3.1.1 XEF_load_custom_file 100
3.1.2 XEF_load_file 101
3.1.3 XEF_parse_file 102
3.1.4 init_libxm 104

Preface

The audience for this document are software developers that have to use directly the services of XtratuM. The reader is expected to have strong knowledge of the LEON3 (SPARC v8) architecture and experience in programming device drivers. It is also advisable to have some knowledge of the ARINC-653 and 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: core/include/sparcv8/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: core/include/sparcv8/arch_types.h

1.1.3 Return codes

The API provides a return code as a 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: 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 1
#define XM_ABI_SUBVERSION 0
#define XM_ABI_REVISION 0

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

Listing 1.4: 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 interation. */
    xm_u64_t currentMaf;            /* [[OPTIONAL]] */
    /* Total number of system messages: */
    xm_u64_t noSamplingPortMsgsRead; /* [[OPTIONAL]] */
    xm_u64_t noSamplingPortMsgsWritten; /* [[OPTIONAL]] */
    xm_u64_t noQueuingPortMsgsSent; /* [[OPTIONAL]] */
    xm_u64_t noQueuingPortMsgsReceived; /* [[OPTIONAL]] */
    xmTime_t irqLastOccurence[CONFIG_NO_HWIRQS];
#ifdef CONFIG_LEON_EDAC_SUPPORT
    xm_u32_t edacErrorCounter;
#endif
} xmSystemStatus_t;
```

Listing 1.5: 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: 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];
#ifdef CONFIG_LEON_EDAC_SUPPORT
    xm_u32_t edacErrorCounter;
#endif
} xmSystemStatus_t;
```

Listing 1.7: 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;
    xmId_t id;
    // Copy of kthread->ctrl.flags
    xm_u32_t flags;
#define PARTITION_SYSTEM_F (1<<0) // 1:SYSTEM
#define PARTITION_FP_F (1<<1) // Floating point enabled
#define PARTITION_HALTED_F (1<<2) // 1:HALTED
#define PARTITION_SUSPENDED_F (1<<3) // 1:SUSPENDED
#define PARTITION_READY_F (1<<4) // 1:READY
//#define PARTITION_FLUSH_DCACHE_F (1<<5)
//#define PARTITION_FLUSH_ICACHE_F (1<<6)
#define PARTITION_DCACHE_ENABLED_F (1<<7)
#define PARTITION_ICACHE_ENABLED_F (1<<8)
```

```
    xm_u32_t imgStart;
    xm_u32_t hwIrqs; // Hw interrupts belonging to the partition
    xm_s32_t noPhysicalMemAreas;
    xm_s32_t noCommPorts;
    xm_u8_t name[CONFIG_ID_STRING_LENGTH];
    xm_u32_t iFlags; // As defined by the ARCH (ET+PIL in sparc)
    xm_u32_t hwIrqsPend; // pending hw irqs
    xm_u32_t hwIrqsMask; // masked hw irqs

    xm_u32_t extIrqsPend; // pending extended irqs
    xm_u32_t extIrqsMask; // masked extended irqs
    struct pctArch arch;
    struct {
        xm_u32_t noSlot:16, releasePoint:1, reserved:15;
        xm_u32_t id;
        xm_u32_t slotDuration;
    } schedInfo;
    xm_u16_t trap2Vector[NO_TRAPS];
    xm_u16_t hwIrq2Vector[CONFIG_NO_HWIRQS];
    xm_u16_t extIrq2Vector[XM_VT_EXT_MAX];
} partitionControlTable_t;
```

Listing 1.8: core/include/guest.h

1.3.3 Memory Map Table (MMT)

The API provides de 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: 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: 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: core/include/hypercalls.h

1.6 Plan management type

1.6.1 Plan management type

The API rovides 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: 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: 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: 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.
    //xm_u32_t flags;
} xmQueuingPortStatus_t;
```

Listing 1.15: 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: 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_u16_t signature;
//   xm_u16_t checksum;
    xm_u32_t opCode;
#define HMLOG_OPCODE_EVENT_MASK (0x1fff<<HMLOG_OPCODE_EVENT_BIT)
#define HMLOG_OPCODE_EVENT_BIT 19
// Bits 18 and 17 free
#define HMLOG_OPCODE_SYS_MASK (0x1<<HMLOG_OPCODE_SYS_BIT)
#define HMLOG_OPCODE_SYS_BIT 16
#define HMLOG_OPCODE_VALID_CPUCTXT_MASK (0x1<<
    HMLOG_OPCODE_VALID_CPUCTXT_BIT)
#define HMLOG_OPCODE_VALID_CPUCTXT_BIT 15
#define HMLOG_OPCODE_MODID_MASK (0x7f<<HMLOG_OPCODE_MODID_BIT)
#define HMLOG_OPCODE_MODID_BIT 8
#define HMLOG_OPCODE_PARTID_MASK (0xff<<HMLOG_OPCODE_PARTID_BIT)
#define HMLOG_OPCODE_PARTID_BIT 0
//   xm_u32_t seq;
    xmTime_t timestamp;
    union {
#define XM_HMLOG_PAYLOAD_LENGTH 4
        struct hmCpuCtxt cpuCtxt;
        xmWord_t payload[XM_HMLOG_PAYLOAD_LENGTH];
    };
}  __PACKED;

typedef struct xmHmLog xmHmLog_t;
```

Listing 1.17: core/include/objects/hm.h

The status type is defined as:

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

Listing 1.18: 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: 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_MODID_SW_ERROR 0
  #define XM_HM_MODID_HWTIMER_ERROR 1

#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_EV_EXTSYNC_ERROR 10
```

Listing 1.20: core/include/xmconf.h

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

```
#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 1.21: core/include/sparcv8/irqs.h

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

```
#define XM_HM_EV_APP_DEADLINE_MISSED (XM_HM_MAX_GENERIC_EVENTS+
    XM_HM_MAX_ARCH_EVENTS+0)
#define XM_HM_EV_APP_APPLICATION_ERROR (XM_HM_MAX_GENERIC_EVENTS+
    XM_HM_MAX_ARCH_EVENTS+1)
#define XM_HM_EV_APP_NUMERIC_ERROR (XM_HM_MAX_GENERIC_EVENTS+
    XM_HM_MAX_ARCH_EVENTS+2)
#define XM_HM_EV_APP_ILLEGAL_REQUEST (XM_HM_MAX_GENERIC_EVENTS+
    XM_HM_MAX_ARCH_EVENTS+3)
#define XM_HM_EV_APP_STACK_OVERFLOW (XM_HM_MAX_GENERIC_EVENTS+
    XM_HM_MAX_ARCH_EVENTS+4)
#define XM_HM_EV_APP_MEMORY_VIOLATION (XM_HM_MAX_GENERIC_EVENTS+
    XM_HM_MAX_ARCH_EVENTS+5)
#define XM_HM_EV_APP_HARDWARE_FAULT (XM_HM_MAX_GENERIC_EVENTS+
    XM_HM_MAX_ARCH_EVENTS+6)
#define XM_HM_EV_APP_POWER_FAIL (XM_HM_MAX_GENERIC_EVENTS+
    XM_HM_MAX_ARCH_EVENTS+7)
```

Listing 1.22: core/include/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 1.23: core/include/objects/trace.h

The status type is defined as:

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

Listing 1.24: core/include/objects/trace.h

1.9.2 Trace events symbols

Additionally, the following symbols for XtratuM events' traces are provided:

```
#define XM_AUDIT_START_WATCHDOG 0x1
#define XM_AUDIT_END_WATCHDOG 0x2
#define XM_AUDIT_BEGIN_PARTITION 0x3
#define XM_AUDIT_END_PARTITION 0x4
#define XM_AUDIT_BEGIN_CS 0x5
#define XM_AUDIT_END_CS 0x6
#define XM_AUDIT_BEGIN_IDLE 0x7
#define XM_AUDIT_END_IDLE 0x8
```

Listing 1.25: core/include/audit.h

1.10 Interrupt management types

1.10.1 Interrupt management symbols

The following interrupt types are provided:

```
   #define XM_TRAP_TYPE 0x0
   #define XM_HWIRQ_TYPE 0x1
   #define XM_EXTIRQ_TYPE 0x2
```

Listing 1.26: core/include/hypercalls.h

The API provides the following symbols for interrupts:

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

#ifdef CONFIG_LEON3FT

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

#else
```

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

#endif

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

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

#define XM_VT_EXT_CYCLIC_SLOT_START (8+XM_VT_EXT_FIRST)

#define XM_VT_EXT_MEM_PROTECT   (16+XM_VT_EXT_FIRST)

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

Listing 1.27: core/include/guest.h

The following hardware exception list is available:

```
#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 1.28: core/include/sparcv8/irqs.h

The handled interrupts list is:

```
#if defined(CONFIG_LEON2)|| defined(CONFIG_LEON3)

#define INTERNAL_BUS_TRAP_NR 1
#ifdef CONFIG_LEON2
#define UART2_TRAP_NR 2
#define UART1_TRAP_NR 3
#endif
#ifdef CONFIG_LEON3
#define UART2_TRAP_NR 3
#define UART1_TRAP_NR 2
#endif
#define IO_IRQ0_TRAP_NR 4
#define IO_IRQ1_TRAP_NR 5
#define IO_IRQ2_TRAP_NR 6
#define IO_IRQ3_TRAP_NR 7
#define TIMER1_TRAP_NR 8
#define TIMER2_TRAP_NR 9
#define DSU_TRAP_NR 11
#define PCI_TRAP_NR 14

#elif defined(CONFIG_LEON3FT)

  #define INTERNAL_BUS_TRAP_NR 1
  #define UART1_TRAP_NR 2
  #define IO_IRQ0_TRAP_NR 3
  #define IO_IRQ1_TRAP_NR 4
  #define UART2_TRAP_NR 17
  #define UART3_TRAP_NR 18
  #define UART4_TRAP_NR 19
  #define UART5_TRAP_NR 20
  #define UART6_TRAP_NR 21
  #define TIMER_LATCH_TRAP_NR 7
  #define TIMER1_TRAP_NR 8
  #define TIMER2_TRAP_NR 9
  #define TIMER3_TRAP_NR 10
  #define TIMER4_TRAP_NR 11
  #define PCI_TRAP_NR 14

#endif
```

Listing 1.29: core/include/sparcv8/irqs.h

1.10.2 Mapping of interrupts to traps

Standard SparcV8 interrupts:

```
#define INTERRUPT_LEVEL_15 0x1f // UNUSED // 15
#define INTERRUPT_LEVEL_14 0x1e // PCI // 14
#define INTERRUPT_LEVEL_13 0x1d // UNUSED // 13
#define INTERRUPT_LEVEL_12 0x1c // UNUSED // 12
#define INTERRUPT_LEVEL_11 0x1b // DSU // 11
#define INTERRUPT_LEVEL_10 0x1a // UNUSED // 10
#define INTERRUPT_LEVEL_9 0x19 // TIMER2 // 9
#define INTERRUPT_LEVEL_8 0x18 // TIMER1 // 8
#define INTERRUPT_LEVEL_7 0x17 // IO3 // 7
#define INTERRUPT_LEVEL_6 0x16 // IO2 // 6
#define INTERRUPT_LEVEL_5 0x15 // IO1 // 5
#define INTERRUPT_LEVEL_4 0x14 // IO0 // 4
#define INTERRUPT_LEVEL_3 0x13 // UART1 // 3 //LEON3 -> UART2
#define INTERRUPT_LEVEL_2 0x12 // UART2 // 2 //LEON3 -> UART1
#define INTERRUPT_LEVEL_1 0x11 // AMBA // 1
```

Listing 1.30: core/include/sparcv8/irqs.h

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: 20] .

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: 18]

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. Partitions are warm reset.

XM_COLD_RESET:
: XtratuM provides two different behaviours depending on whether the *Jump to user function on cold reset system* option was selected during the compilation of the hypervisor or not. When this option is not selected, XtratuM unconditionally jumps back to the resident software entry point. This entry point is provided as part of the configuration of the system, being its validity not checked by XtratuM.

  On the other hand, when this option is set, XtratuM requires to the user to implement a function (void `UsrReset(xmHmLog_t *log)`) which is compiled and binded within XtratuM. In this case, XtratuM calls this function.

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: 17]

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.

History:

Added in version 3.1.2

Usage examples:

```
struct xmPhysicalMemMap *xmPM;
...
xmPM = XM_get_partition_mmap();
```

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: 16] .

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 theirselves. 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: 25] , `XM_suspend_partition` [Page: 29] ,

`XM_resume_partition` [Page: 27] .

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 `Point` 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 (defined in the XML configuration 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: 21] , XM_suspend_partition [Page: 29] ,
XM_resume_partition [Page: 27] , XM_memory_copy [Page: 56]

2.2.7 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: 29] .

2.2.8 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: 21] 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 theirselves. 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: 25] , `XM_suspend_partition` [Page: 29] ,

`XM_resume_partition` [Page: 27] .

2.2.9 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 theirselves. 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: 25] , `XM_halt_partition` [Page: 21] ,

`XM_resume_partition` [Page: 27] .

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: 32] .

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: 31] .

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: 36]

2.4.2 XM_reload_watchdog

Synopsis: Set the system watchdog bugdet.

Category:

System partition service.

Declaration:

```
xm_s32_t XM_reload_watchdog (
/*in */    xmTime_t  budget);
```

Description:

If budget is zero, then the system watchdog is set as expired. Otherwise, the watchdog expires when the clock reaches the value specified by the budget parameter plus the clock value when the hypercall is called.

Meanwhile the watchdog is not expired, a predefined action is done each partition context switch. For more information read XtratuM User Manual [section 5.13].

Return value:

[XM_OK]
Successful completion.

[XM_PERM_ERROR]
The calling partition is not a system partition.

[XM_INVALID_PARAM]
The budget parameter has a negative value.

Rationale:

Internally, the system watchdog is implemented as a timer. When the timer expired the watchdog is set as expired.

Usage examples:

```
#define WD_BUDGET 1000000ULL /* 1 second */

XM_reload_watchdog(WD_BUDGET);
```

2.4.3 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 absolut 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 disables, 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-programed 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 LEON board), the one shot timer mode is the best technique because it provides a high resolution and a 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: 33]

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: 39] .

2.5.2 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: 38] .

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: 52] , `XM_receive_queuing_message` [Page: 50] , `XM_get_queuing_port_status` [Page: 45] .

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: 54] , `XM_read_sampling_message` [Page: 48] , `XM_get_sampling_port_status` [Page: 47] .

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: 52] , `XM_receive_queuing_message` [Page: 50] ,

`XM_create_queuing_port` [Page: 40] , `XM_get_queuing_port_status` [Page: 45] .

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: 52] , `XM_receive_queuing_message` [Page: 50] , `XM_create_queuing_port` [Page: 40] .

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]

Succesful 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: 54] , XM_read_sampling_message [Page: 48] ,

XM_create_sampling_port [Page: 42] , XM_get_sampling_port_status [Page: 47] .

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: 54] , `XM_read_sampling_message` [Page: 48] ,
`XM_create_sampling_port` [Page: 42]

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: 42] , XM_write_sampling_message [Page: 54] , XM_get_sampling_port_status [Page: 47] .

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 meesage 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: 40] , XM_send_queuing_message [Page: 52] , XM_get_queuing_port_status [Page: 45] , XM_get_queuing_port_status [Page: 45] .

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: 40] , XM_receive_queuing_message [Page: 50] , XM_get_queuing_port_status [Page: 45] , XM_get_queuing_port_status [Page: 45] .

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 automically 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: 42] , `XM_read_sampling_message` [Page: 48] , `XM_get_sampling_port_status` [Page: 47] .

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_SYSTEM_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 `destId` partition, or the source area does not belong to the `srcId` partition.
- The destination is a read-only area.

- The dstAddr and/or srcAddr are not 4 bytes aligned.

[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]`
Successful completion.

`[XM_INVALID_PARAM]`
`cacheType` is not a valid cache type.
`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. The errors that can be raised through this service are listed in section 1.8.3.

HM event
XM_HM_EV_APP_DEADLINE_MISSED
XM_HM_EV_APP_APPLICATION_ERROR
XM_HM_EV_APP_NUMERIC_ERROR
XM_HM_EV_APP_ILLEGAL_REQUEST
XM_HM_EV_APP_STACK_OVERFLOW
XM_HM_EV_APP_MEMORY_VIOLATION
XM_HM_EV_APP_HARDWARE_FAULT
XM_HM_EV_APP_POWER_FAIL

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]
Successful completion.

[XM_INVALID_PARAM]
The event param is not a valid application HM event.

History:

Introduced in XtratuM 3.9.0.

See also:

XM_hm_read [Page: 60] , XM_hm_status [Page: 61] .

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 prtition.
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: 59] , XM_hm_status [Page: 61] .

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: 60] , `XM_hm_status` [Page: 61]

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: 63] , `XM_trace_status` [Page: 64] .

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: 62] , `XM_trace_status` [Page: 64] .

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: 62] , `XM_trace_read` [Page: 63] .

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 (hardware and extended ones) 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 `hwIrqsMask` and `extIrqMask` parameters.

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

Return value:

[XM_OK]
Successful completion.

Rationale:

This function replaces the deprecated `XM_unmask_irq()`.

History:

Added in version 3.1.2

Usage examples:

```
    XM_clear_irqmask(0, (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
    //unmasks the HW_TIMER interrupt.

    XM_clear_irqmask(0xFFFFFFFF, 0);
    //This call unmasks all hardware interrupts and leaves the
        extended
    //mask as it is.
```

See also:

`XM_set_irqmask` [Page: 71] , `XM_set_irqpend` [Page: 72] , `XM_clear_irqpend` [Page: 66] .

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 (hardware and extended ones) 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 `hwIrqsPend` and `extIrqPend` parameters are cleared.

Return value:

[XM_OK]
Successful completion.

History:

Added in version 3.1.2

Usage examples:

```
        XM_clear_irqpend(0, (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.

        XM_clear_irqpend(0xFFFFFFFF, 0);
        //This call clears all hardware interrupts an leaves the
            extended
        //pending interrupts as they are.
```

See also:

XM_set_irqpend [Page: 72] , XM_set_irqmask [Page: 71] , XM_clear_irqmask [Page: 65] .

2.10.3 XM_disable_irqs

Synopsis: Disable interrupts.

Category:

Standard service.

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: The hypercall always succeeds, not returning any value.

History:

Added in version 3.1.2

See also:

`XM_set_irqmask` [Page: 71] , `XM_set_irqpend` [Page: 72] , `XM_clear_irqpend` [Page: 66] , `XM_enable_irqs` [Page: 68] .

2.10.4 XM_enable_irqs

Synopsis: Enable interrupts.

Category:

Standard service.

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: The hypercall always succeeds. No value is returned.

History:

Added in version 3.1.2

See also:

XM_set_irqmask [Page: 71] , XM_set_irqpend [Page: 72] , XM_clear_irqpend [Page: 66] , XM_disable_irqs [Page: 67] .

2.10.5 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 an 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_IPVI0);
    // partition 1 and 2 will receive the ipvi XM_VT_EXT_IPVI0 //
    ...
    rtn=XM_raise_ipvi(XM_VT_EXT_IPVI1);
    // partition 2 will receive the ipvi XM_VT_EXT_IPVI1 //
    ...
  XM_halt_partition(XM_PARTITION_SELF);
}
```

2.10.6 XM_route_irq

Synopsis: Link an interrupt with the vector generated when the interrupt is issued.

Category:

Standard service

Declaration:

```
xm_s32_t XM_route_irq (
/*in */    xm_u32_t  type,
/*in */    xm_u32_t  irq,
/*in */    xm_u16_t  vector);
```

Description:

The `XM_route_irq()` hypercall binds an exception/hw-interrupt/extended-interrupt (given by the combination `type`: XM_TRAP_TYPE, XM_HWIRQ_TYPE, XM_EXTIRQ_TYPE and the number of interrupt) with an interrupt vector.

An interrupt vector is the index into the interrupt vector table. When an interrupt is triggered, this index is used by the hypervisor together with the base address of the interrupt vector table to jump to the interrupt handler.

Return value:

[XM_OK]
Successful completion.

[XM_INVALID_PARAM]
`type` is not a valid interrupt type (XM_TRAP_TYPE, XM_HWIRQ_TYPE or XM_EXT_IRQ).
`irq` is not a valid interrupt number.
`vector` is not a valid index to the interrupt vector table.

Usage examples:

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

void PartitionMain(void) {
  xm_u32_t flags;

  xprintf("\n+++ route: %x\n", XM_params_get_PCT()->hwIrq2Vector[0]);
  XM_route_irq(XM_HWIRQ_TYPE, 0, 2);
  xprintf("\n+++ route: %x\n", XM_params_get_PCT()->hwIrq2Vector[0]);
  xprintf("\n+++ pending: %x\n", XM_params_get_PCT()->hwIrqsPend);
  XM_set_irqpend(0x6, 0x0);
  xprintf("\n+++ pending: %x\n", XM_params_get_PCT()->hwIrqsPend);
  XM_clear_irqmask(0x16, 0x0);
  flags=XM_sparc_get_psr();
  flags=~(0xf00);
  XM_sparc_set_psr(flags);
  flags=XM_sparc_get_psr();
  xprintf("flags %x\n", flags);
  while(1);

  XM_halt_partition(XM_PARTITION_SELF);
}
```

2.10.7 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) hardware and extended interrupts. The interrupt lines set in the `hwIrqsMask` and `extIrqMask` parameters are masked.

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

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(0, 0xffffffff);
//this call leaves the HW_IRQ mask as it is and masks all
    extended interrupts.
```

See also:

`XM_clear_irqmask` [Page: 65]

2.10.8 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 `extIrqPend` 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.

Return value:

[XM_OK]
Successful completion.

History:

Added in version 3.1.2

Usage examples:

```
    XM_set_irq_pend(0,~0x1);
    //This call leaves the pending HW_IRQ's as they are and sets
    //all EXT_IRQ's as pending.
```

See also:

`XM_clear_irqpend` [Page: 66] ,, `XM_set_irqmask` [Page: 71] , `XM_clear_irqmask` [Page: 65] .

2.11 Miscelaneous

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 funcion 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 funcion 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 Sparcv8 specific

2.12.1 XM_sparc_clear_pil

Synopsis: Clear the PIL field of the PSR (enable interrupts).

Category:

Standard service. Assembly hypercall.

Declaration:

```
void XM_sparc_clear_pil(void);
```

Description:

Clear the PIL field of the PSR register (bits 8 to 11) to the value 0x0.

As a result of this operation, interrupts will be attended by the processor.

Return value:

This hypercall always succeeds.

History:

Added in version 3.1.2

Usage examples:

```
XM_sparc_clear_pil();
```

See also:

`XM_clear_irqmask` [Page: 65] , `XM_set_irqmask` [Page: 71] ,
`XM_sparc_set_pil` [Page: 87] .

2.12.2 XM_sparc_ctrl_winflow

Synopsis: Check the state of register windows.

Category:

Standard service. Assembly hypercall.

Declaration:

```
void XM_sparc_ctrl_winflow(void);
```

Description:

Checks the state of register windows in order to prevent a windows underflow.

Avoids windows underflow by shifting the register windows upward.

The hypercall is provided as an assembly macro.

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

Return value:

No value is returned.

Rationale:

This function could be needed in complex interrupt handlers such as the ones found in RTEMS. In these OS the interrupts handler are usually written in assembly code which makes it sometimes necessary to manually manage the register windows.

Usage examples:

```
XM_sparc_ctrl_winflow():
```

2.12.3 XM_sparc_disable_sdp

Synopsis: Disable the SDP to enable single-byte writing in EEPROM.

Category:

Standard service.

Declaration:

```
xm_s32_t XM_sparc_disable_sdp (
/*in */    xm_u32_t  chipSelect);
```

Description:

This service allows a partition with part of the EEPROM allocated to disable SDP. It is not advised to use this feature. If it must be used, it is advisable to allocate the whole chip of EEPROM in the XM_CF, since **it may interfere other partitions' operation if the same chip is shared**.

Return value:

[XM_OK]
SDP has been successfully disabled.

[XM_INVALID_PARAM]
`chipSelect` is out of range.

[XM_PERMISSION_ERROR]
The partition has not access to the `chipSelect` port according to XM_CF.

See also: `XM_sparc_outport_sdp` [Page: 86] , `XM_sparc_inport` [Page: 80] , `XM_sparc_outport` [Page: 82] .

2.12.4 XM_sparc_flush_regwin

Synopsis: Save the contents of the register window.

Category:

Standard service. Assembly hypercall.

Declaration:

```
void XM_sparc_flush_regwin(void);
```

Description:

Saves the contents of all the register windows in use except for the active window 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.

Usage examples:

```
XM_sparc_flush_regwin();
```

2.12.5 XM_sparc_get_psr

Synopsis: Get the ICC, EF and PIL flags from the virtual PSR processor register.

Category:

Standard service. Assembly hypercall.

Declaration:

```
xm_u32_t XM_sparc_get_psr(void);
```

Description:

Returns ICC (Integer Condition Codes) fields, EF (Enable Floating-Point) field and PIL (Processor Interrupt Level) fields of the PSR (Processor Status Register). The bits 20 to 23 contain the ICC flags; bits 8 to 11 contain the PIL; bit 12 contains the EF.

Return value: The value returned by this hypercall contains the aforementioned ICC and PIL flags of the PSR.

Rationale:

This function replaces the deprecated `XM_sparc_get_flags()`.

History:

Added in version 3.1.2

Usage examples:

```
xm_u32_t psr;
psr = XM_sparc_get_psr();
```

See also:

`XM_sparc_set_psr` [Page: 88] .

2.12.6 XM_sparc_inport

Synopsis: Read from a hardware I/O port.

Category:

Standard service.

Declaration:

```
xm_s32_t XM_sparc_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]
Successful completion.

[XM_INVALID_PARAM]

- `portAddr` is not aligned to 4 bytes,
- `ptrValue` address does not belong to the address space of the calling partition.

[XM_PERM_ERROR]
`portAddr` is not allocated to the calling partition,

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.

Usage examples:

```
#include "std_c.h"
#include &/>

#define LEON_BASE 0x80000000
#define UART2_BASE 0x80
#define UART2_CTRL 0x8

void PartitionMain(void) {
  xm_u32_t value;

  if (XM_sparc_inport(LEON_BASE+UART2_BASE+UART2_CTRL, &amp;value)==
      XM_OK)
        xprintf("[Before] UART2 ctrl: 0x%x\n", value);
  else
        xprintf("Unable to read from 0x%x\n", LEON_BASE+UART2_BASE+
            UART2_CTRL);
  xprintf("Writting 0xf in 0x%x (UART2 ctrl)\n",
  LEON_BASE+UART2_BASE+UART2_CTRL);
  xprintf("ret: %d\n", XM_sparc_outport(LEON_BASE+UART2_BASE+
      UART2_CTRL,
  0xf));
  if (XM_sparc_inport(LEON_BASE+UART2_BASE+UART2_CTRL, &amp;value)==
      XM_OK)
        xprintf("[After] UART2 ctrl: 0x%x\n", value);
  else
        xprintf("Unable to read from 0x%x\n", LEON_BASE+UART2_BASE+
            UART2_CTRL);
  xprintf("Writting 0x10 in 0x%x (not allowed port)\n",
  LEON_BASE+UART2_BASE+UART2_CTRL+0x10);
  xprintf("ret: %d\n",
  XM_sparc_outport(LEON_BASE+UART2_BASE+UART2_CTRL+0x10, 0xf));
  XM_halt_partition(XM_PARTITION_SELF);
}
```

See also: `XM_sparc_outport` [Page: 82] , `XM_memory_copy` [Page: 56]

2.12.7 XM_sparc_outport

Synopsis: Write in a hardware I/O port.

Category:

Standard service.

Declaration:

```
xm_s32_t XM_sparc_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="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. 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&gt:
            <Restricted address="0x3000" mask="0xff" />
        </IoPorts>
    </HwResources>
</Partition>
```

Return value:

[XM_OK]
The value has been successfully written.

[XM_INVALID_PARAM]
portAddr is not aligned to 4 bytes,

[XM_PERM_ERROR]
portAddr is not allocated to the calling partition,

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.

Usage examples:

```
#include "std_c.h"
#include &/>

#define LEON_BASE 0x80000000
#define UART2_BASE 0x80
#define UART2_CTRL 0x8

void PartitionMain(void) {
  xm_u32_t value;

  if (XM_sparc_inport(LEON_BASE+UART2_BASE+UART2_CTRL, &amp;value)==
      XM_OK)
        xprintf("[Before] UART2 ctrl: 0x%x\n", value);
  else
        xprintf("Unable to read from 0x%x\n", LEON_BASE+UART2_BASE+
            UART2_CTRL);

  xprintf("Writting 0xf in 0x%x (UART2 ctrl)\n",
  LEON_BASE+UART2_BASE+UART2_CTRL);
  xprintf("ret: %d\n", XM_sparc_outport(LEON_BASE+UART2_BASE+
      UART2_CTRL,
  0xf));
  if (XM_sparc_inport(LEON_BASE+UART2_BASE+UART2_CTRL, &amp;value)==
      XM_OK)
        xprintf("[After] UART2 ctrl: 0x%x\n", value);
  else
        xprintf("Unable to read from 0x%x\n", LEON_BASE+UART2_BASE+
            UART2_CTRL);
  xprintf("Writting 0x10 in 0x%x (not allowed port)\n",
  LEON_BASE+UART2_BASE+UART2_CTRL+0x10);
  xprintf("ret: %d\n",
  XM_sparc_outport(LEON_BASE+UART2_BASE+UART2_CTRL+0x10, 0xf));
  XM_halt_partition(XM_PARTITION_SELF);
}
```

See also: XM_sparc_inport [Page: 80] , XM_memory_copy [Page: 56]

2.12.8 XM_sparc_outport_msg

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

Category:

Standard service.

Declaration:

```
xm_s32_t XM_sparc_outport_msg(
/*in */    xm_u32_t                portAddr,
/*in */    struct ioOutPortMsg * ioMsg,
/*in */    xm_u32_t                timeout);
```

Description:

This service provides direct hardware access to the peripherals. On response to this call, XtratuM writes the message contained in the struct pointed by `ioMsg` into the port `portAddr` on the native hardware.

The parameter `ioMsg` is a struct defined as follows:

```
struct ioOutPortMsg{
  xm_u32_t * msg;
  xm_s32_t size;
  xmAddress_t checkingAddr;
  xm_u32_t validationValue;
  xm_u32_t validationMask;
};
```

where `msg` is a pointer to the message to be written, `size` is the length of the message, `checkingAddr` is the address used to validate that the data has been written, `validationValue` is the value to check in `checkingAddr`, and `validationMask` contains the bits that will be taken into account in `checkingAddr` and `validationValue`. Note that if `validationMask` is equal to 0, any value on `checkingAddr` will validate that the data has been written.

The parameter `timeout` indicates the maximum time waited on each 32 bytes written.

The configuration file lists the hardware ports that can be used by each partition. For more information on how to allocate ports to a partition in the configuration file see `XM_sparc_outport`.

Return value:

[XM_OK]
The message has been successfully written.

[XM_INVALID_PARAM]
`portAddr` it is not aligned to 4 bytes.
`ioMsg` or `ioMsg->msg` is not a valid address.

[XM_PERM_ERROR]
`portAddr` or `ioMsg->checkingAddr` is not allocated to this partition.

[XM_OP_NOT_ALLOWED]
Timeout has been reached.

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.

Usage examples:

```
#include "std_c.h"
#include &/>

#define LEON_BASE 0x80000000
#define UART2_BASE 0x80
#define UART2_CTRL 0x8

void PartitionMain(void) {
  xm_u32_t value;
  xm_u32_t msg[5] = {0xf0, 0xf1, 0xf2, 0xf3, 0xf4};
  struct ioOutPortMsg ioMsg;
  xm_u32_t timeout = 100LL;

  if (XM_sparc_inport(LEON_BASE+UART2_BASE+UART2_CTRL, &amp;value)==
      XM_OK)
        xprintf("[Before] UART2 ctrl: 0x%x\n", value);
  else
        xprintf("Unable to read from 0x%x\n", LEON_BASE+UART2_BASE+
            UART2_CTRL);

  xprintf("Writing ioMsg->msg in 0x%x (UART2 ctrl)\n",
  LEON_BASE+UART2_BASE+UART2_CTRL);
  xprintf("ret: %d\n", XM_sparc_outport_msg(LEON_BASE+UART2_BASE+
      UART2_CTRL,
  &ioMsg, timeout));
  if (XM_sparc_inport(LEON_BASE+UART2_BASE+UART2_CTRL, &amp;value) ==
       XM_OK)
        xprintf("[After] UART2 ctrl: 0x%x\n", value);
  else
        xprintf("Unable to read from 0x%x\n", LEON_BASE+UART2_BASE+
            UART2_CTRL);
  xprintf("Writing ioMsg->msg in 0x%x (not allowed port)\n",
  LEON_BASE+UART2_BASE+UART2_CTRL+0x10);
  xprintf("ret: %d\n",
  XM_sparc_outport_msg(LEON_BASE+UART2_BASE+UART2_CTRL+0x10, &ioMsg,
       timeout));
  XM_halt_partition(XM_PARTITION_SELF);
}
```

See also: XM_sparc_inport [Page: 80] , XM_sparc_outport [Page: 82]

2.12.9 XM_sparc_outport_sdp

Synopsis: Write in EEPROM atomically or enable SDP mode.

Category:

Standard service.

Declaration:

```
xm_s32_t XM_sparc_outport_sdp (
/* out */ xm_u8_t *dst,
/* in */ xm_u8_t  *src,
/* in */ xm_u32_t size);
```

Description:

This service provides direct hardware access to EEPROM. On response to this call, XtratuM performs the write of `size` bytes from `src` into `dst` on the native hardware. EEPROM is treated in Xtratum as a special peripheral.

If the parameter `size` is equal to zero, SDP is enabled and no further changes are produced on EEPROM.

Return value:

[XM_OK]
Successful completion.

[XM_PERM_ERROR]
The partition has not access to the port `dst` according to the XM_CF file.

[XM_INVALID_PARAM] The size parameter value is out of range or dst address is out of range.

See also: `XM_sparc_disable_sdp` [Page: 77] , `XM_sparc_inport` [Page: 80] , `XM_sparc_outport` [Page: 82] .

2.12.10 XM_sparc_set_pil

Synopsis: Set the PIL field of the PSR (disallow interrupts).

Category:

Standard service. Assembly hypercall.

Declaration:

```
void XM_sparc_set_pil(void);
```

Description:

Set the PIL field of the PSR register (bit [11:8]) to the value 0xF.

As a result of this operation, interrupts will no longer be attended by the processor. Note that interrupt level 15 is always attended (non-disable, non-maskable).

Return value:

This hypercall always succeeds.

History:

Added in version 3.1.2

Usage examples:

```
XM_sparc_set_pil();
```

See also:

`XM_clear_irqmask` [Page: 65] , `XM_set_irqmask` [Page: 71] ,

`XM_sparc_clear_pil` [Page: 75]

2.12.11 XM_sparc_set_psr

Synopsis: Set the ICC, EF and PIL flags on the virtual PSR processor register.

Category:

Standard service. Assembly hypercall.

Declaration:

```
void XM_sparc_set_psr(
/*in */    xm_u32_t  flags);
```

Description:

Sets ICC (Integer Condition Codes, bits 20 to 23) fields and PIL (Processor Interrupt Level, bits 8 to 11)fields of the PSR according to the `flags` parameter. The EF (Enable Floating-Point, bit 12) field is set if the partition is allowed to use the FPU, otherwise the value of this bit is discarded.

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

Rationale:

This function replaces the deprecated `XM_sparc_set_flags()`.

History:

Added in version 3.1.2

Usage examples:

```
    XM_set_psr(~0x0);
```

See also:

`XM_sparc_get_psr` [Page: 79] .

2.12.12 sparc_iret_nr

Synopsis: Return from a trap.

Category:

Standard service. Assembly hypercall.

Declaration:

```
sparc_iret_nr
```

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 by corrupted.

The ICC fields of the PSR are restored with the content of the register `l0`.

Given its nature, there is no C interface provided for this call. 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. No value is returned.

Rationale:

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

- The register `l0` contains a copy of the ICC flags.
- The register `l1` contains the PC (program counter).
- The register `l2` contains the nPC (next program counter).

Usage examples:

```
exception_handler_asm:
       mov sparc_get_psr_nr, %o0
       __XM_AHC
       mov %o0, %l0
       sub %fp, 48, %fp
       std %l0, [%fp+40]
       std %l2, [%fp+32]
       std %g6, [%fp+24]
       std %g4, [%fp+16]
       std %g2, [%fp+8]
       st %g1, [%fp+4]
       rd %y, %g5
       st %g5, [%fp]
       mov %l5, %o0
       call exception_handler
       sub %fp, 0x80, %sp
       ld [%fp], %g1
       wr %g1, %y
       ld [%fp+4], %g1
       ldd [%fp+8], %g2
```

```
        ldd [%fp+16], %g4
        ldd [%fp+24], %g6
        ldd [%fp+32], %l2
        ldd [%fp+40], %l0
        add %fp, 48, %fp
        mov %l0, %o1
        mov sparc_set_psr_nr, %o0
        __XM_AHC
        set sparc_iret_nr, %o0
        __XM_AHC


/* 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, %l5 ;\
        nop
```

2.12.13 sparc_write_tbr_nr

Synopsis: Write TBR value.

Category:

Standard service. Assembly hypercall.

Declaration:

```
sparc_write_tbr_nr
/*in */    xmWord_t   val
```

Description:

Updates the virtualised Trap Base Register with the `val` value provided as a parameter.

Given its nature, there is no C interface provided for this call. The hypercall is provided as an assembly macro.

Return value:

No value is returned.

Usage examples:

```
    set sparc_write_tbr_nr, %o0
    set traptab_address, %o1
    __XM_HC
```

2.13 Obsolete

2.13.1 XM_sparc_flush_cache

Synopsis: Flush data cache. Assembly hypercall.

Category:

Standard service.

Declaration:

```
void XM_sparc_flush_cache(void);
```

Description:

Invalidates the data cache. All the cache lines are marked as empty.

Return value:

This hypercall always succeeds.

History:

Added in version 3.1.2

Usage examples:

```
XM_sparc_flush_cache();
```

2.14 Library functions

2.14.1 XEF_load_custom_file

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

Category:

Library service.

Declaration:

```
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:

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

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:

```
...
struct xefOps xefOps;
struct xefFile xefFile;
struct xefCustomFile xefCustomFile;

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

See also:

`XEF_parse_file` [Page: 102] , `XEF_load_file` [Page: 101]

2.14.2 XEF_load_file

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

Category:

Library service.

Declaration:

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

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.

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);
}
```

See also:

`XEF_parse_file` [Page: 102] , `XEF_load_custom_file` [Page: 100]

2.14.3 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.

Usage examples:

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

See also:

XEF_load_file [Page: 101] , XEF_load_custom_file [Page: 100]

2.14.4 init_libxm

Synopsis: Performs libXM parameters initialisation.

Declaration:

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

Description:

The `init_libxm` performs initialisation tasks for libXM. It stores a refence 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.

This page is intentionally left blank.

Chapter 3

Library functions

3.1 XEF functions

3.1.1 XEF_load_custom_file

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

Category:

Library service.

Declaration:

```
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:

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

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:

```
...
struct xefOps xefOps;
struct xefFile xefFile;
struct xefCustomFile xefCustomFile;

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

See also:

`XEF_parse_file` [Page: 102] , `XEF_load_file` [Page: 101]

3.1.2 XEF_load_file

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

Category:

Library service.

Declaration:

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

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.

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);
}
```

See also:

`XEF_parse_file` [Page: 102] , `XEF_load_custom_file` [Page: 100]

3.1.3 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.

Usage examples:

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

See also:

XEF_load_file [Page: 101] , XEF_load_custom_file [Page: 100]

3.1.4 init_libxm

Synopsis: Performs libXM parameters initialisation.

Declaration:

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

Description:

The `init_libxm` performs initialisation tasks for libXM. It stores a refence 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.