XtratuM Hypervisor for x86

Reference Manual

Fent Innovative Software Solutions

July, 2013
Reference: fnts-xm-rm-42b

This page is intentionally left blank.

Contents

This page is intentionally left blank.

Preface

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

Typographical conventions

The following font conventions are used in this document:

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

Caution sign

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

Acknowledgements

This work has been performed in the frame of the project “Open and cost-effective virtualization techniques and supporting separation kernel for the embedded systems industry” (VOS4ES).

This page is intentionally left blank.

Chapter 1

Introduction

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

1.1 XtratuM overview

Synopsis: Global data structures and types.

Global data types:

```
// Basic types
typedef unsigned char xm_u8_t;
typedef char xm_s8_t;
typedef unsigned short xm_u16_t;
typedef short xm_s16_t;
typedef unsigned int xm_u32_t;
typedef int xm_s32_t;
typedef unsigned long long xm_u64_t;
typedef long long xm_s64_t;
```

Listing 1.1: /core/include/ia32/arch_types.h

```
// Extended types
typedef xm_s64_t xmTime_t;
typedef xm_u32_t xmAddress_t;
typedef xm_u16_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/ia32/arch_types.h

API and ABI version numbers:

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

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

Listing 1.3: /core/include/hypercalls.h

Error codes:

```
#define XM_OK              (0)
#define XM_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 1.4: /core/include/hypercalls.h

Interrupts:

Native hardware interrupts (interrupts generated by the real hardware) are received by the partition in the same way than in the host machine. The first native interrupt causes the trap number 0x20. This number can be remapped through the Partition Control Table.

Extended interrupts are raised in response to XtratuM events. The new interrupts set are in the range from 32 to 63, and causes traps from 244 to 255 respectively.

Exceptions:

List of exceptions (traps triggered by the processor):

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

Listing 1.5: /core/include/ia32/irqs.h

Processor exceptions are managed through the health monitoring system.

Partition Control Table (PCT):

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

    xmAtomic_t iFlags;
    // BIT: 23..16: ARCH
    //      1: TRAP PENDING
    //      0: IRQ
    xmAtomic_t hwIrqsPend; // pending hw irqs
    xmAtomic_t hwIrqsMask; // masked hw irqs

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

    xmAtomic_t objDescClassPend; // Object descritors

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

Listing 1.6: /core/include/guest.h

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

Listing 1.7: /core/include/ia32/guest.h

1.2 Health Monitoring overview

Synopsis: Health Monitoring overview.

Health monitoring events:

There are tree groups of hm events:

Processor triggered

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

```
#define XM_HM_EV_IA32_DIVIDE_EXCEPTION      (XM_HM_MAX_GENERIC_EVENTS + 0)
#define XM_HM_EV_IA32_DEBUGGER_EXCEPTION    (XM_HM_MAX_GENERIC_EVENTS + 1)
#define XM_HM_EV_IA32_NMI_EXCEPTION         (XM_HM_MAX_GENERIC_EVENTS + 2)
#define XM_HM_EV_IA32_BREAKPOINT_EXCEPTION  (XM_HM_MAX_GENERIC_EVENTS + 3)
#define XM_HM_EV_IA32_OVERFLOW_EXCEPTION    (XM_HM_MAX_GENERIC_EVENTS + 4)
#define XM_HM_EV_IA32_BOUNDS_EXCEPTION      (XM_HM_MAX_GENERIC_EVENTS + 5)
#define XM_HM_EV_IA32_INVALID_OPCODE        (XM_HM_MAX_GENERIC_EVENTS + 6)
#define XM_HM_EV_IA32_COPROCESOR_UNAVAILABLE (XM_HM_MAX_GENERIC_EVENTS + 7)
#define XM_HM_EV_IA32_DOUBLE_FAULT          (XM_HM_MAX_GENERIC_EVENTS + 8)
#define XM_HM_EV_IA32_COPROCESSOR_OVERRUN   (XM_HM_MAX_GENERIC_EVENTS + 9)
#define XM_HM_EV_IA32_INVALID_TSS           (XM_HM_MAX_GENERIC_EVENTS + 10)
#define XM_HM_EV_IA32_SEGMENT_NOT_PRESENT   (XM_HM_MAX_GENERIC_EVENTS + 11)
#define XM_HM_EV_IA32_STACK_FAULT           (XM_HM_MAX_GENERIC_EVENTS + 12)
#define XM_HM_EV_IA32_GENERAL_PROTECTION_FAULT (XM_HM_MAX_GENERIC_EVENTS +
    13)
#define XM_HM_EV_IA32_PAGE_FAULT            (XM_HM_MAX_GENERIC_EVENTS + 14)
#define XM_HM_EV_IA32_RESERVED              (XM_HM_MAX_GENERIC_EVENTS + 15)
#define XM_HM_EV_IA32_MATH_FAULT            (XM_HM_MAX_GENERIC_EVENTS + 16)
#define XM_HM_EV_IA32_ALIGNMENT_CHECK       (XM_HM_MAX_GENERIC_EVENTS + 17)
#define XM_HM_EV_IA32_MACHINE_CHECK         (XM_HM_MAX_GENERIC_EVENTS + 18)
#define XM_HM_EV_IA32_FLOATING_POINT_EXCEPTION (XM_HM_MAX_GENERIC_EVENTS +
    19)
```

Listing 1.8: /core/include/ia32/xmconf.h

User triggered

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

XtratuM triggered

```
#define XM_HM_EV_INTERNAL_ERROR 0
#define XM_HM_EV_UNEXPECTED_TRAP 1
#define XM_HM_EV_PARTITION_UNRECOVERABLE 2
#define XM_HM_EV_PARTITION_ERROR 3
#define XM_HM_EV_PARTITION_INTEGRITY 4
#define XM_HM_EV_MEM_PROTECTION 5

#define XM_HM_EV_OVERRUN 6

#define XM_HM_EV_SCHED_ERROR 7
#define XM_HM_EV_WATCHDOG_TIMER 8

#define XM_HM_EV_INCOMPATIBLE_INTERFACE 9
```

Listing 1.9: /core/include/xmconf.h

Default actions:

The following tables summarises the list of health monitoring events. The default action (marked with a $\star$ symbol). The allowed actions (those that can be selected in the XM_CF file) are marked with the $\bullet$ symbol. The last column indicates if the event is logger or not (if it is nit explicitly configured in the XM_CF file).

If this HM event is propagated to the partition, the trap number raised is the same than the one of the native hardware, and not the HM event number. Note that **XM_HM_EV_*** constants do not match the trap numbers (see Listing 1.5).

	XM_HM_AC_COLD_RESET	XM_HM_AC_HALT	XM_HM_AC_IGNORE	XM_HM_AC_PROPAGATE	XM_HM_AC_SHUTDOWN	XM_HM_AC_SUSPEND	XM_HM_AC_WARM_RESET	Log event
XM_HM_EV_IA32_ALIGNMENT_CHECK	•	•	•	⋆	•	•	•	No
XM_HM_EV_IA32_BOUNDS_EXCEPTION	•	•	•	⋆	•	•	•	No
XM_HM_EV_IA32_BREAKPOINT_EXCEPTION	•	•	•	⋆	•	•	•	No
XM_HM_EV_IA32_COPROCESOR_UNAVAILABLE	•	•	•	⋆	•	•	•	No
XM_HM_EV_IA32_COPROCESSOR_OVERRUN	•	•	•	⋆	•	•	•	No
XM_HM_EV_IA32_DEBUGGER_EXCEPTION	•	•	•	⋆	•	•	•	No
XM_HM_EV_IA32_DIVIDE_EXCEPTION	•	•	•	⋆	•	•	•	No
XM_HM_EV_IA32_DOUBLE_FAULT	•	•	•	⋆	•	•	•	No
XM_HM_EV_IA32_FLOATING_POINT_EXCEPTION	•	•	•	⋆	•	•	•	No
XM_HM_EV_IA32_GENERAL_PROTECTION_FAULT	•	•	•	⋆	•	•	•	No
XM_HM_EV_IA32_INVALID_OPCODE	•	•	•	⋆	•	•	•	No
XM_HM_EV_IA32_INVALID_TSS	•	•	•	⋆	•	•	•	No
XM_HM_EV_IA32_MACHINE_CHECK	•	•	•	⋆	•	•	•	No
XM_HM_EV_IA32_MATH_FAULT	•	•	•	⋆	•	•	•	No
XM_HM_EV_IA32_NMI_EXCEPTION	•	•	•	⋆	•	•	•	No
XM_HM_EV_IA32_OVERFLOW_EXCEPTION	•	•	•	⋆	•	•	•	No
XM_HM_EV_IA32_PAGE_FAULT	•	•	•	⋆	•	•	•	No
XM_HM_EV_IA32_RESERVED	•	•	•	⋆	•	•	•	No
XM_HM_EV_IA32_SEGMENT_NOT_PRESENT	•	•	•	⋆	•	•	•	No
XM_HM_EV_IA32_STACK_FAULT	•	•	•	⋆	•	•	•	No
XM_HM_EV_INCOMPATIBLE_INTERFACE	•	•	•		•	•	•	No
XM_HM_EV_INTERNAL_ERROR	•	•	•		•	•	•	No
XM_HM_EV_MEM_PROTECTION	•	•	•		•	•	•	No
XM_HM_EV_OVERRUN	•	•	•		•	•	•	No
XM_HM_EV_PARTITION_ERROR	•	•	•		•	•	•	No
XM_HM_EV_PARTITION_INTEGRITY	•	•	•		•	•	•	No
XM_HM_EV_PARTITION_UNRECOVERABLE	•	•	•		•	•	•	No
XM_HM_EV_SCHED_ERROR	•	•	•		•	•	•	No
XM_HM_EV_UNEXPECTED_TRAP	•	•	•		•	•	•	No
XM_HM_EV_WATCHDOG_TIMER	•	•	•		•	•	•	No

Table 1.1: Partition scope HM events

	XM_HM_AC_COLD_RESET	XM_HM_AC_HALT	XM_HM_AC_IGNORE	XM_HM_AC_PROPAGATE	XM_HM_AC_SHUTDOWN	XM_HM_AC_SUSPEND	XM_HM_AC_WARM_RESET	Log event
XM_HM_EV_IA32_ALIGNMENT_CHECK	•	⋆	•	•	•	•	•	Yes
XM_HM_EV_IA32_BOUNDS_EXCEPTION	•	⋆	•	•	•	•	•	Yes
XM_HM_EV_IA32_BREAKPOINT_EXCEPTION	•	⋆	•	•	•	•	•	Yes
XM_HM_EV_IA32_COPROCESOR_UNAVAILABLE	•	⋆	•	•	•	•	•	Yes
XM_HM_EV_IA32_COPROCESSOR_OVERRUN	•	⋆	•	•	•	•	•	Yes
XM_HM_EV_IA32_DEBUGGER_EXCEPTION	•	⋆	•	•	•	•	•	Yes
XM_HM_EV_IA32_DIVIDE_EXCEPTION	•	⋆	•	•	•	•	•	Yes
XM_HM_EV_IA32_DOUBLE_FAULT	•	⋆	•	•	•	•	•	Yes
XM_HM_EV_IA32_FLOATING_POINT_EXCEPTION	•	⋆	•	•	•	•	•	Yes
XM_HM_EV_IA32_GENERAL_PROTECTION_FAULT	•	⋆	•	•	•	•	•	Yes
XM_HM_EV_IA32_INVALID_OPCODE	•	⋆	•	•	•	•	•	Yes
XM_HM_EV_IA32_INVALID_TSS	•	⋆	•	•	•	•	•	Yes
XM_HM_EV_IA32_MACHINE_CHECK	•	⋆	•	•	•	•	•	Yes
XM_HM_EV_IA32_MATH_FAULT	•	⋆	•	•	•	•	•	Yes
XM_HM_EV_IA32_NMI_EXCEPTION	•	⋆	•	•	•	•	•	Yes
XM_HM_EV_IA32_OVERFLOW_EXCEPTION	•	⋆	•	•	•	•	•	Yes
XM_HM_EV_IA32_PAGE_FAULT	•	⋆	•	•	•	•	•	Yes
XM_HM_EV_IA32_RESERVED	•	⋆	•	•	•	•	•	Yes
XM_HM_EV_IA32_SEGMENT_NOT_PRESENT	•	⋆	•	•	•	•	•	Yes
XM_HM_EV_IA32_STACK_FAULT	•	⋆	•	•	•	•	•	Yes
XM_HM_EV_INCOMPATIBLE_INTERFACE	•	•	•		•	•	•	No
XM_HM_EV_INTERNAL_ERROR	•	•	•				•	No
XM_HM_EV_MEM_PROTECTION	•	•	•				•	No
XM_HM_EV_OVERRUN	•	•	•				•	No
XM_HM_EV_PARTITION_ERROR	•	•	•				•	No
XM_HM_EV_PARTITION_INTEGRITY	•	•	•				•	No
XM_HM_EV_PARTITION_UNRECOVERABLE	•	•	•				•	No
XM_HM_EV_SCHED_ERROR	•	•	•				•	No
XM_HM_EV_UNEXPECTED_TRAP	•	•	•				•	No
XM_HM_EV_WATCHDOG_TIMER	•	•	•		•	•	•	No

Table 1.2: System scope HM events

Health monitoring actions:

```
#define XM_HM_AC_IGNORE 0
#define XM_HM_AC_SHUTDOWN 1
#define XM_HM_AC_COLD_RESET 2
#define XM_HM_AC_WARM_RESET 3
#define XM_HM_AC_SUSPEND 4
#define XM_HM_AC_HALT 5
#define XM_HM_AC_PROPAGATE 6
```

Listing 1.10: /core/include/xmconf.h

[XM_HM_AC_IGNORE] No action is performed.

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

[XM_HM_AC_COLD_RESET] The offending partition or the hypervisor is cold reset.

[XM_HM_AC_WARM_RESET] The offending partition or the hypervisor is warm reset.

[XM_HM_AC_SUSPEND] The offending partition is suspended.

[XM_HM_AC_HALT] The offending partition or the hypervisor is halted.

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

[XM_HM_AC_SWITCH_TO_MAINTENANCE] A synchronous plan switch to maintenance plan (Plan 2) is performed. That is, the current plan is terminated and the plan 2 is immediately started.

Health monitoring logs:

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

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

Listing 1.11: /core/include/objects/hm.h

See also:

XM_hm_open [Page: 26], XM_hm_read [Page: 27], XM_hm_seek [Page: 28], XM_hm_status [Page: 30]

Chapter 2

Hypercalls

2.1 XM_are_irqs_enabled

Synopsis: Checks if interrupts are enabled.

Category:

Standard service.

Declaration:

```
xm_s32_t XM_are_irqs_enabled(void);
```

Description:

The `XM_are_irqs_enabled()` hypercall checks if interrupts are enabled by checking the `iFlag` field of the `PCT`.

Return value:

[0] if interrupts are disabled.

[1] if interrupts are enabled.

2.2 XM_create_queuing_port

Synopsis: Create a queuing port.

Category:

Standard service.

Declaration:

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

Description:

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

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

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

Return value:

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

[XM_INVALID_CONFIG]

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

Usage examples:

```
xm_s32_t portDesc;

portDesc = XM_create_queuing_port ("example",
                                   30,
                                   XM_DESTINATION_PORT);
if (portDesc < 0) {
    XM_write_console("Port example could not be created\n",34);
    XM_halt_partition(0,-1);
}
```

See also:

XM_send_queuing_message [Page: 58], XM_receive_queuing_message [Page: 52],

XM_get_queuing_port_status [Page: 21].

2.3 XM_create_sampling_port

Synopsis: Create a sampling port.

Category:

Standard service.

Declaration:

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

Description:

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

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

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

Return value:

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

[XM_INVALID_CONFIG]

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

Usage examples:

```
xm_s32_t portDesc;

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

See also:

XM_write_sampling_message [Page: 81], XM_read_sampling_message [Page: 50],

XM_get_sampling_port_status [Page: 22].

2.4 XM_ctrl_object

Synopsis: Performs a control operation on a object.

Category:

Standard service.

Declaration:

```
xm_s32_t XM_ctrl_object(xmObjDesc_t objDesc,
                        xm_u32_t cmd, void *arg);
```

Description:

The `XM_ctrl_object()` hypercall performs the control `cmd` command with `arg` as arguments on the `objDesc` object. TBD

Return value:

[`XM_INVALID_PARAM`] Is returned if the `objDesc` is not valid.

[`XM_OK`] The operation succeeded.

See also:

`XM_read_object` [Page: 49], `XM_write_object` [Page: 79], `XM_seek_object` [Page: 57]

2.5 XM_disable_irqs

Synopsis: Disable processor irqs.

Declaration:

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

Description:

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

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

Return value:

This hypercall always succeed.

See also:

XM_unmask_irq [Page: 75], XM_mask_irq [Page: 42],

XM_enable_irqs [Page: 15]

2.6 XM_enable_irqs

Synopsis: Enable processor irqs.

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 are disabled, then the highest priority interrupt will then be received by the partition when interrupts are enabled.

Return value:

This hypercall always succeed.

See also:

`XM_unmask_irq` [Page: 75], `XM_mask_irq` [Page: 42],

`XM_disable_irqs` [Page: 14]

2.7 XM_exec_pendirqs

Synopsis: Executes any pending hardware or extended interrupts.

Category:

Standard service.

Declaration:

```
xm_s32_t XM_exec_pendirqs(void *ctxt);
```

Description:

The `XM_exec_pendirqs()` hypercall checks the `iFlags` field and emulates the occurrence of the pending interrupts.

Return value:

[0] No pending interrupts emulated.

[1] Pending interrupts emulated.

2.8 XM_flush_hyp_batch

Synopsis: Flush a batch of multiple hypercalls.

Category:

Standard service.

Declaration:

```
xm_s32_t XM_flush_hyp_batch(void);
```

Description:

The `XM_flush_hyp_batch()` executes any "lazy" hypercalls which have been requested by `XM_lazy_hypercall()`.

See also:

`XM_lazy_hypercall` [Page: 36]

2.9 XM_get_partition_status

Synopsis: Get the current status of a partition.

Category:

Standard service.

Declaration:

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

Description:

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

The `xmPartitionStatus_t` is a data type contains the following fields:

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

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

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

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

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

Return value:

[XM_OK]
The operation succeeded.

[XM_PERM_ERROR]
The calling partition is not supervisor.

[XM_INVALID_PARAM]
`id` is not a valid identifier, or `state` is not a valid partition address.

History:

Introduced in XtratuM 2.2.0.

See also:

`XM_system_get_status` [Page: 68].

2.10 XM_get_physmem_map

Synopsis: Returns the physical memory map of the partition.

Category:

Standard service.

Declaration:

```
xm_s32_t XM_get_physmem_map(struct xmcMemoryArea *memMap,
                            xm_s32_t noAreas);
```

Description:

The XM_get_physmem_map() hypercall copies noAreas memory areas from the memory map of the calling partition to the struct xmcMemoryArea struct pointed by memMap.

```
#define XM_PCT_SLOT_CYCLIC 0x0
#define XM_PCT_SLOT_SPARE 0x1
#endif

struct xmcMemoryArea
    xmAddress_t startAddr;
    xmSize_t size;
#define XM_MEM_AREA_SHARED (1<<0)
#define XM_MEM_AREA_MAPPED (1<<1)
#define XM_MEM_AREA_WRITE (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)
```

Listing 2.2: /core/include/xmconf.h

Return value:

The return value of the XM_get_physmem_map() hypercall is the the number of memory areas contained in the partition memory map.

In case of error the following codes are returned:

[XM_INVALID_PARAM] if the memMap is out of range.

[XM_INVALID_PARAM] if the memMap is not big enough to hold the memory areas.

[XM_INVALID_PARAM] if the noAreas is out of range.

Rationale:

With this table, the partition know how much memory, and where it is initially mapped. It can also translate virtual to physical addresses.

Usage examples:

```
struct xmcMemoryArea memAreas[xmPartInfTab->
    noPhysicalMemoryAreas];
int n = XM_get_physmem_map(memAreas, xmPartInfTab->
    noPhysicalMemoryAreas);
if (n != xmPartInfTab->noPhysicalMemoryAreas)
        return -1;
```

2.11 XM_get_plan_status

Synopsis: Return information about the scheduling plans.

Category:

Standard service.

Declaration:

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

Description:

Returns in `state` information about the previous, the current and the next plan.

The `xmPlanStatus_t` is a data type which contains the following fields:

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

Listing 2.3: /core/include/objects/status.h

[switchTime]
The absolute time of the last plan switch request. After a reset (both warm and cold), the value is set to zero.

[current]
Identifier of the current plan.

[next]
The plan identifier that will be active on the next major frame. If no plan switch is going to occur, then the value of `next` is equal to the value of `current`.

[prev]
The identifier of the plan executed before the current one. After a reset (both warm and cold) the value is set to (-1).

Return value:

[XM_OK]
The operation succeeded.

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

[XM_INVALID_PARAM]
`state` is not a valid partition address.

History:

Introduced in XtratuM 3.1.2.

See also:

`XM_set_plan` [Page: 62].

2.12 XM_get_queuing_port_status

Synopsis: Get the status of a queuing port.

Category:

Standard service.

Declaration:

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

Description:

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

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

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

Return value:

The return values are:

[XM_OK]
: The operation succeeds.

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

See also:

XM_send_queuing_message [Page: 58], XM_receive_queuing_message [Page: 52], XM_create_queuing_port [Page: 11].

2.13 XM_get_sampling_port_status

Synopsis: Get the status of a sampling port.

Category:

Standard service.

Declaration:

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

Description:

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

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

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

Return value:

[XM_OK]
The operation succeeds.

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

See also:

XM_write_sampling_message [Page: 81], XM_read_sampling_message [Page: 50], XM_create_sampling_port [Page: 12]

2.14 XM_get_time

Synopsis: Retrieve the time of the specified clock.

Category:

Standard service.

Declaration:

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

Description:

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

XtratuM provides the next clocks:

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

Listing 2.6: /core/include/hypercalls.h

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

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

Return value:

[XM_OK]
The operation succeeds.

[XM_INVALID_PARAM]
If the `clockId` is a non valid virtual clock.

Rationale:

Since the resolution of the clock is 1 microsecond it may happen that the same time value is returned if the function is called twice rapidly.

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

Usage examples:

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

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

See also: XM_set_timer [Page: 64]

2.15 XM_halt_partition

Synopsis: Terminates a partition.

Category:

Standard service/Supervisor reserved.

Declaration:

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

Description:

The XM_halt_partition() hypercall causes the termination of the partitionId partition. The partition is set in halt state.

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

Only supervisor partitions can halt other partitions than itself. Any partition can halt itself. The constant XM_PARTITION_SELF represents the calling partition.

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

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

Return value:

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

[XM_OK]
: The target partition has been successfully halted, or the target partition was already int halt state.

[XM_INVALID_PARAM]
: partitionId is not a valid partition identifier.

[XM_PERM_ERROR]
: The calling partition is not supervisor and the target partition is not itself.

Usage examples:

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

See also:

XM_reset_partition [Page: 53], XM_suspend_partition [Page: 67],

XM_resume_partition [Page: 56].

2.16 XM_halt_system

Synopsis: Stop the system.

Category:

Supervisor reserved.

Declaration:

```
xm_s32_t XM_halt_system();
```

Description:

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

This function shall be used with extreme caution. Only a hardware reset can reboot the system. If the watchdog is armed, when the watchdog timer will expire the board will restart.

This is a supervisor reserved function.

Return value:

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

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

Rationale:

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

See also:

XM_reset_system [Page: 55]

2.17 XM_hm_open

Synopsis: Open the health monitoring log stream.

Category:

Standard service.

Declaration:

```
xm_s32_t XM_hm_open ();
```

Description:

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

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

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

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

Return value:

[XM_OK]
The operation succeeded.

[XM_PERM_ERROR]
The calling partition is not supervisor.

History:

Introduced in XtratuM 2.2.0.

See also:

XM_hm_read [Page: 27], XM_hm_seek [Page: 28], XM_hm_status [Page: 30]

2.18 XM_hm_read

Synopsis: Read a health monitoring log entry.

Category:

Standard service.

Declaration:

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

Description:

Attempts to read one health monitoring log entry from the current read position into the structure pointed by `hmLogPtr`. If no log entry is available, the call returns immediately.

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

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

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

Return value:

[0] There are no new log entries in the HM log stream. No data is returned.

[1]
The next log entry is returned and the current read position is increased by one.

[XM_PERM_ERROR]
The calling partition is not supervisor.

[XM_INVALID_PARAM]
The `hmLogPtr` address is not a valid partition address.

History:

Introduced in XtratuM 2.2.0.

Usage examples:

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

See also:

`XM_hm_open` [Page: 26], `XM_hm_seek` [Page: 28], `XM_hm_status` [Page: 30]

2.19 XM_hm_seek

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

Category:

Standard service.

Declaration:

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

Description:

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

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

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

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

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

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

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

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

Return value:

[XM_OK]
The position has been set.

[XM_PERM_ERROR]
The calling partition is not supervisor.

[XM_INVALID_PARAM]
The `offset` or `whence` are not valid.

History:

Introduced in XtratuM 2.2.0.

Usage examples:

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

See also:

XM_hm_open [Page: 26], XM_hm_read [Page: 27], XM_hm_status [Page: 30]

2.20 XM_hm_status

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

Category:

Standard service.

Declaration:

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

Description:

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

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

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

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

.

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

Return value:

[XM_OK]
The operation succeeded.

[XM_PERM_ERROR]
The calling partition is not supervisor.

[XM_INVALID_PARAM]
The hmStatusPtr address is not a valid partition address.

History:

Introduced in XtratuM 2.2.0.

See also:

XM_hm_open [Page: 26], XM_hm_read [Page: 27], XM_hm_seek [Page: 28]

2.21 XM_ia32_set_idt_desc

Synopsis: Sets an Interrupt Descriptor Table entry.

Category:

Standard service.

Declaration:

```
xm_s32_t XM_ia32_set_idt_desc(xm_s32_t entry,
                              genericDesc_t *desc);
```

Return value:

[XM_INVALID_PARAM]

[XM_INVALID_PARAM]

[XM_INVALID_PARAM]

[XM_OK]

2.22 XM_ia32_update_sys_struct

Synopsis: Update a processor control register.

Category:

Standard service.

Declaration:

```
xm_s32_t XM_ia32_update_sys_struct(xm_u32_t procStruct,
                                   xm_u32_t val1, xm_u32_t val2);
```

Description:

This hypercall updates the contents of th `procStruct` with the values provided in `val1 and val2`.

The values that `procStruct` can take are:

```
#define IA32_UPDATE_SSESP1 0
#define IA32_UPDATE_SSESP2 1
#define IA32_UPDATE_GDT 2

#define NR_IA32_PROC_STRUCT 3
```

Listing 2.8: /core/include/arch/processor.h

Return value:

[XM_INVALID_PARAM]

[XM_OK]

2.23 XM_idle_self

Synopsis: Idles the execution of the calling partition.

Category:

Standard service.

Declaration:

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

Description:

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

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

Rationale:

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

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

Usage examples:

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

```
#include <guest.h>

#define HW_IRQS 16

partitionControlTable_t pct;

    ...

    /* Mask all partition hw interrupts */
    for (i=XM_VT_HW_FIRST; i <= HW_VT_HW_LAST; i++) {
        XM_mask_irq(i);
    }
    /* Mask all partition extended interrupts */
    for (i=XM_VT_EXT_FIRST; i <= HW_VT_EXT_LAST; i++) {
        XM_mask_irq(i);
    }
    /* Unmask the start slot interrupt */
    XM_unmask_irq(XM_VT_EXT_CYCLIC_SLOT_START);
    XM_enable_irqs();
    /* Wait the start of the next MAF */
    if (pct.tag == 0) { /* If we are in the first slot of
                           the current MAF, then skip it. */
        XM_idle_self();
    }
    while (pct.tag != 0) {
        XM_idle_self(); /* Skip all the non-starting slots */
```

```
}
/* Implementation of a cyclic plan to run partition tasks: */
while (1) {
    TaskA(); TaskB();
    XM_idle_self(); /* Wait for the first slot. */
    TaskC(); TaskB();
    XM_idle_self(); /* Wait for the second slot. */
    TaskD(); TaskE(); TaskB();
    XM_idle_self(); /* Wait for the third slot. */
}
XM_write_console("PANIC: unreachable code\n",25)
XM_halt_partition(XM_PARTITION_SELF);

...
```

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

2.24 XM_iret

Synopsis: Return from a interrupt.

Category:

Standard service. Assembly hypercall.

Declaration:

```
XM_iret(void);
```

Description:

Emulates a `iret` (Interrupt RET) assembly instruction.

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

The hypercall is provided as an assembly macro.

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

Usage examples:

```
CommonTrapBody:
        HW_SAVE_ALL
        pushl 44(%esp)
    call ExceptionHandler
    addl $4, %esp
        HW_RESTORE_ALL
        jmp XM_iret

#define BUILD_TRAP_ERRCODE(trapnr) \
    .section .rodata.trapHndl,"a" ; \
    .align 4 ; \
    .long TrapHndl##trapnr ; \
    .text ; \
    .align 4 ; \
    TrapHndl##trapnr: ; \
        pushl $trapnr ; /* error_code has already been filled */ \
        jmp CommonTrapBody
```

2.25 XM_lazy_hypercall

Synopsis: Execute a sequence of hypercalls.

Category:

Standard service.

Declaration:

```
void XM_lazy_hypercall(xm_u32_t noHyp, xm_s32_t noArgs, ...);
```

Description:

Calling the XM_flush_hyp_batch() hypercall is not mandatory, the XM_lazy_hypercall takes care of XM_flush_hyp_batch() This function requests the hypercall to be performed in a

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

See also:

XM_flush_hyp_batch [Page: 17]

2.26 XM_lazy_ia32_update_sys_struct

Synopsis:

Category:

Standard service.

Declaration:

```
void XM_lazy_ia32_update_sys_struct(xm_u32_t procStruct,
                                    xm_u32_t val1,
                                    xm_u32_t val2);
```

Description:

where register to modify can be one of the following:

[IA32_UPDATE_SSESP1]

[IA32_UPDATE_SSESP2]

[IA32_UPDATE_GDT]

2.27 XM_lazy_set_page_type

Synopsis: Set the type of a page.

Category:

Standard service.

Declaration:

```
void XM_lazy_set_page_type(xmAddress_t pAddr, xm_u32_t type);
```

2.28 XM_lazy_update_page32

Synopsis:

Category:

Standard service.

Declaration:

```
void XM_lazy_update_page32(xmAddress_t pAddr, xm_u32_t val);
```

2.29 XM_lazy_write_register32

Synopsis: Modify a processor control register of 32-bit width.

Category:

Standard service.

Declaration:

```
void XM_lazy_write_register32(xm_u32_t reg32, xm_u32_t val);
```

2.30 XM_lazy_write_register64

Synopsis: Modify a processor control register of 64-bit width.

Category:

Standard service.

Declaration:

```
void XM_lazy_write_register64(xm_u32_t reg64,
                              xm_u32_t valH, xm_u32_t valL);
```

2.31 XM_mask_irq

Synopsis: Mask interrupt.

Declaration:

```
xm_s32_t XM_mask_irq(xm_u32_t noIrq)
```

Description:

This function masks (blocks) the selected hardware or extended interrupt.

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

Return value:

[XM_OK]
The operation succeeded.

[XM_INVALID_PARAM]
If noIrq is invalid.

See also:

XM_unmask_irq [Page: 75].

2.32 XM_memory_copy

Synopsis: Copy copies data from/to address spaces.

Category:

Standard service/Supervisor reserved.

Declaration:

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

Description:

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

This function copies `size` bytes from the area pointed by `srcAddr` located in the address space of `srcId` partition to the address `destAddr` in the memory space of `destId` partition.

The following considerations shall be taken into account:

- The source and destination areas shall not overlap to avoid data corruption.
- When copying from memory to memory, for efficiency reasons, both areas shall be 8 bytes aligned.
- Since this function allows to copy large blocks of memory, care must be taken to avoid breaking temporal isolation.
- This function has been implemented using one object descriptor (associated with the `/dev/mem` object). The maximum number of object descriptors per partition is a source code configuration parameter, and shall be large enough.

- If the source or destination addresses do not belong to the space of a partition (for example, ROM areas) then the `XM_SYSTEM_ID` shall be used.

  Only supervisor partitions are allowed to perform a copy for/to other address space than its own (i.e. other partitions or system space).
- The source or/and destination are mapped I/O registers, then the I/O mapped addresses shall be word (4 bytes) aligned.
- Note that the `size` parameter indicates the number of **bytes** to be copied, and an I/O register is 4 bytes.
- Only memory mapped I/O registers fully allocated to the partition (that is, using the `Range` element in the `XM_CF` configuration file) can be addressed by the `XM_memory_copy` function.

Return value:

`[XM_OK]`
The operation succeeds.

`[XM_INVALID_PARAMS]`

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

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

[XM_PERM_ERROR]
The calling partition is not supervisor.

Rationale:

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

Partition identifiers are not checked in the current implementation.

2.33 XM_multicall

Synopsis: Execute a sequence of hypercalls.

Category:

Standard service. Optimization. [Not ported to LEON2]

Declaration:

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

Description:

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

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

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

Return value: TBD.

2.34 XM_override_trap_hndl

Synopsis: Override a trap handler entry.

Category:

Standard service.

Declaration:

```
xm_s32_t XM_override_trap_hndl(xm_s32_t entry,
                               struct trapHandler *);
```

Return value:

[XM_INVALID_PARAM]

[XM_INVALID_PARAM]

[XM_OK]

2.35 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.

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

    xmAtomic_t iFlags;
    // BIT: 23..16: ARCH
    //      1: TRAP PENDING
    //      0: IRQ
    xmAtomic_t hwIrqsPend; // pending hw irqs
    xmAtomic_t hwIrqsMask; // masked hw irqs

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

    xmAtomic_t objDescClassPend; // Object descritors

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

Listing 2.9: /core/include/guest.h

Return value: This function always succeeds.

History:

Added in version 3.1.2

2.36 XM_raise_ipvi

Synopsis: Raise an extended interrupt.

Category:

Standard service.

Declaration:

```
xm_s32_t XM_raise_ipvi(xm_u32_t partition_id, xm_u8_t no_ipvi);
```

Description:

This hypercall raises a IPVI (Inter Processor Virtual Interrupt) number no_ipvi to the partition with id paritition_id.

Return value:

[XM_INVALID_PARAM]

- The partition_id is out of range.
- The no_ipvi is out of range.

[XM_OK] The operation suceeded.

Usage examples:

```
XM_raise_ipvi(1, XM_VT_EXT_IPVI0);
```

2.37 XM_read_object

Synopsis: Performs a read on a object.

Category:

Standard service.

Declaration:

```
xm_s32_t XM_read_object(xmObjDesc_t objDesc, void *buffer,
    xm_u32_t size, xm_u32_t *flags);
```

Description:

This hypercall reads `size` bytes from the `objDesc` object and writes them on the address pointed by `buffer`.

Return value:

[XM_INVALID_PARAM]

- The `buffer` is out of range.
- The `size` is out of range.
- `objDesc` is not a valid object descriptor or the `objDesc` object has `read` disabled.

See also:

`XM_write_object` [Page: 79], `XM_seek_object` [Page: 57], `XM_ctrl_object` [Page: 13]

2.38 XM_read_sampling_message

Synopsis: Reads a message from the specified sampling port.

Category:

Standard service.

Declaration:

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

Description:

The `XM_read_sampling_message()` service is used to read a message from the specified sampling port. If succeed, at most `size` bytes are copied into the buffer pointed by `msgPtr`.

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

Return value:

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

[XM_INVALID_PARAM]

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

[XM_INVALID_CONFIG]

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

Optimization:

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

Usage examples:

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

xm_u32_t  flags;


portDesc = XM_create_sampling_port ("sample",
                                    30,
                                    XM_PORT_DESTINATION);
if (portDesc < 0) {
    XM_halt_partition(XM_PARTITION_SELF);
}

ret_code = XM_read_sampling_message (portDesc, message, 30, &flags);
```

```
if (ret_code < 0) {
    XM_write_console("Error reading sampling port\n",28);
    return;
}

if (! (flags & XM_MSG_VALID)) {
    XM_write_console("The message is not valid!\n",27);
    return;
}

message[29]=0x0; /* For safety */
XM_write_console(message,ret_code);
```

See also:

XM_create_sampling_port [Page: 12], XM_write_sampling_message [Page: 81], XM_get_sampling_port_status [Page: 22], .

2.39 XM_receive_queuing_message

Synopsis: Receive a message from the specified queuing port.

Category:

Standard service.

Declaration:

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

Description:

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

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

Return value:

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

[XM_NOT_AVAILABLE]
: The channel is empty or the port is not connected to a channel in the `XM_CF` configuration file.

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

[XM_INVALID_CONFIG]
: The `size` is not compatible with the configuration of the specified port.

Optimization:

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

See also:

`XM_create_queuing_port` [Page: 11], `XM_send_queuing_message` [Page: 58],

`XM_get_queuing_port_status` [Page: 21], `XM_get_queuing_port_status` [Page: 21].

2.40 XM_reset_partition

Synopsis: Reset a partition.

Category:

Standard service/Supervisor reserved.

Declaration:

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

Description:

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

`XM_WARM_RESET`:

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

`XM_COLD_RESET`:

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

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

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

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

Return value:

If the target partition (`partitionId`) is the calling partition then 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]`
A non supervisor partition attempted to reset other partition.

Usage examples:

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

See also:

XM_halt_partition [Page: 24], XM_suspend_partition [Page: 67], XM_resume_partition [Page: 56], XM_memory_copy [Page: 43]

2.41 XM_reset_system

Synopsis: Reset the system.

Category:

Supervisor reserved.

Declaration:

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

Description:

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

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

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

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

Return value:

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

[XM_PERM_ERROR]
: The calling partition is not a supervisor partition.

[XM_INVALID_PARAM]
: `mode` is not a valid mode.

Rationale:

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

See also:

`XM_halt_system` [Page: 25]

2.42 XM_resume_partition

Synopsis: Resume the execution of a partition.

Category:

Supervisor reserved.

Declaration:

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

Description:

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

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

Only supervisor partitions can invoke this service.

Return value:

[XM_OK]
The target partition has been resumed.

[XM_INVALID_PARAM]
`partitionId` is not a valid partition identifier.

[XM_PERM_ERROR]
The calling partition does not have supervisor rights.

See also:

XM_suspend_partition [Page: 67].

2.43 XM_seek_object

Synopsis: Performs a seek on a object.

Category:

Standard service.

Declaration:

```
xm_s32_t XM_seek_object(xmObjDesc_t objDesc,
                        xm_u32_t offset, xm_u32_t whence);
```

Return value:

[objectTab[class]->Seek(objDesc,]

[offset,]

[whence)]

[XM_INVALID_PARAM]

2.44 XM_send_queuing_message

Synopsis: Send a message in the specified queuing port.

Category:

Standard service.

Declaration:

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

Description:

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

Return value:

[XM_OK]
The message has been successfully written into the port.

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

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

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

Optimization:

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

See also:

XM_create_queuing_port [Page: 11], XM_receive_queuing_message [Page: 52],
XM_get_queuing_port_status [Page: 21], XM_get_queuing_port_status [Page: 21].

2.45 XM_set_page_type

Synopsis: Changes the type of the physical page pAddr to type.

Category:

Standard service.

Declaration:

```
xm_s32_t XM_set_page_type(xmAddress_t pAddr, xm_u32_t type);
```

Description:

Partitions are able to declare new page tables and use them. In order to perform this operation, the type of the page selected to act as part of a page table must be changed to the appropriate type:

PPAG_STD Standard page, there is no restrictions to the permissions given to this kind of page. They can be mapped as read/write or read only.

PPAG_PTD1 A page of this type will hold a first level page table, any of its entries must either map a PPAG_PTD2 page or not to map anything (not present page). This sort of pages can only be mapped as read-only. This restriction is enforced by XtratuM.

PPAG_PTD2
A page of this type will act as a second level page table, there is no restricions about the pages mapped by this page, however, PPAG_PTD1 and PPAG_PTD2 must be set as read-only pages.

Before change the type of a page, this page must be unmapped from all existing page tables, otherwise, the hypercall returns a XM_OP_NOT_ALLOWED error.

Return value:

[XM_OK] The operation finished successfully.

[XM_INVALID_PARAM]The page pAddr does not belong to the partition or its content breaks some rule for its new type

[XM_OP_NOT_ALLOWED]The page pAddr is mapped in an existing page table.

Usage examples:

```
#include <xm_inc/arch/physmm.h/>
#include <xm.h>

unsigned long pgd[1024] __attribute__((aligned(4096)));
unsigned long pgt[1024] __attribute__((aligned(4096)));
...

void PartitionMain(void) {
   memset(ptdL1, 0, 4096);
   memset(ptdL2, 0, 4096);
   if (XM_set_page_type(ptdL1, PPAG_PTD1)!=XM_OK) {
      xprintf("ptdL1 couldn't be set as PTD1\n");
      XM_halt_partition(XM_PARTITION_SELF);
   }
   if (XM_set_page_type(ptdL2, PPAG_PTD2)!=XM_OK) {
      xprintf("ptdL2 couldn't be set as PTD2\n");
```

```
        XM_halt_partition(XM_PARTITION_SELF);
    }
    if (XM_update_page32(&ptdL1[0], (unsigned long)ptdL2|0x2)!=
        XM_OK) { // PTDL2 PRESENT
        xprintf("ptdL1[0] couldn't be updated\n");
        XM_halt_partition(XM_PARTITION_SELF);
    }
    if (XM_update_page32(&ptdL2[0], 0x40100000|0xe)!=XM_OK) {
        // PAGE PRESENT | USER_RW
        xprintf("ptdL2[0] couldn't be updated\n");
        XM_halt_partition(XM_PARTITION_SELF);
    }

     ...
}
```

See also:

XM_update_page32 [Page: 76]

2.46 XM_set_partition_opmode

Synopsis: Set the Partition Operation mode (TBD)

Declaration:

```
xm_s32_t XM_set_partition_opmode(xm_s32_t opMode);
```

Description:

None

2.47 XM_set_plan

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

Category:

Standard service.

Declaration:

```
xm_s32_t XM_set_plan(xm_u32_t planId);
```

Description:

The plan `planId` is scheduled to be started at the end of the current MAF.

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 causing a system reset.

Return value:

[XM_OK]
The operation succeeded.

[XM_PERM_ERROR]
The calling partition is not system.

[XM_INVALID_PARAM]
`planId` is not a valid plan identifier. Valid plan identifiers are those specified in the `XM_CF` file, except the plan zero.

History:

Introduced in XtratuM 3.1.2.

See also:

`XM_get_plan_status` [Page: 20].

2.48 XM_set_spare_guest

Synopsis: Set spare guest partition.

Category:

Standard service.

Declaration:

```
xm_s32_t XM_set_spare_guest(xm_u32_t partition_id, xmTime_t *
    start, xmTime_t *stop);
```

Description:

This hyercall allows the spare host to schedule the partition with id partition_id. If the call succeeds, the selected partition is scheduled immediately. The partition will be scheduled until it executes `XM_idle_self` [Page: 33], or the current slot ends (the slot assigned to the spare host).

Return value:

[XM_OK]
The operation succeeded.

[XM_PERM_ERROR]
The calling partition is not the spare host.

[XM_INVALID_PARAM]
`partition_id` refers to the spare host.

[XM_INVALID_PARAM]
`partition_id` is not a valid partition identifier.

[XM_INVALID_PARAM]
`partition_id` reters to a partition that is currently in the HALTED state.

History:

Introduced in XtratuM 2.5.1

2.49 XM_set_timer

Synopsis: Arm a timer.

Category:

Standard service.

Declaration:

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

Description:

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

If `interval` is not zero, then the timer will expire periodically at absolut times: `absTime + n * interval`; where ”n” starts in zero and repeats until the timer is re-armed.

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

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

Clock	Associated extended interrupt
XM_HW_CLOCK	XM_VT_EXT_HW_TIMER
XM_EXEC_CLOCK	XM_VT_EXT_EXEC_TIMER

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

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

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

Return value:

`[XM_OK]`

The timer has been successfully armed.

`[XM_INVALID_PARAM]`

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

Rationale:

Internally, the timers are managed in **one shot mode**. That is, the hardware timer is not programmed to generate an interrupt in a periodic way, but it is re-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 LEON2 board), the one shot timer mode is the best technique because it provides an high resolution and small overhead.

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

Usage examples:

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

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

XM_get_time(XM_HW_CLOCK, &Start);
Start += (xmTime_t)1000000;
XM_set_timer(XM_HW_CLOCK, Start, Period);
XM_enable_irqs();
XM_unmask_irq(XM_VT_EXT_HW_TIMER);
```

See also: XM_get_time [Page: 23]

2.50 XM_shutdown_partition

Synopsis: Send a shutdown interrupt to a partition.

Category:

Standard service/Supervisor reserved.

Declaration:

```
xm_s32_t XM_shutdown_partition (xmId_t partitionId);
```

Description:

The XM_shutdown_partition() hypercall raises the extended interrupt XM_VT_EXT_SHUTDOWN on the target partition.

On receiving a shutdown interrupt, the target partition shall close and terminate the ongoing tasks and finally call the XM_halt_partition [Page: 24] hypercall. XtratuM does not control the state the target partition after a shutdown request.

Only supervisor partitions can invoque this service to shutdown other partitions than itself. Any partition can shutdown itself. The constant XM_PARTITION_SELF represents the calling partition.

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

Return value:

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

[XM_OK]
: The shutdown trap has been successfully delivered.

[XM_INVALID_PARAM]
: partitionId is not a valid partition identifier.

[XM_PERM_ERROR]
: The calling partition is not supervisor and the target partition is not itself.

See also:

XM_reset_partition [Page: 53], XM_suspend_partition [Page: 67],

XM_resume_partition [Page: 56].

2.51 XM_suspend_partition

Synopsis: Suspend the execution of a partition.

Category:

Standard service/Supervisor reserved.

Declaration:

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

Description:

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

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

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

Only supervisor partitions can suspend other partitions than itself. Any partition can halt itself. The constant `XM_PARTITION_SELF` represents the calling partition.

Return value:

`[XM_OK]`
The target partition has been suspended or it was already in suspend state.

`[XM_INVALID_PARAM]`
`partitionId` is not a valid partition identifier.

`[XM_PERM_ERROR]`
The calling partition is not supervisor and the target partition is not itself.

See also:

`XM_reset_partition` [Page: 53], `XM_halt_partition` [Page: 24],
`XM_resume_partition` [Page: 56].

2.52 XM_system_get_status

Synopsis: Get the current status of the system.

Category:

Standard service.

Declaration:

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

Description:

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

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

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

Listing 2.10: /core/include/objects/status.h

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

Return value:

[XM_OK]
The operation succeeded.

[XM_PERM_ERROR]
The calling partition is not supervisor.

[XM_INVALID_PARAM]
`state` is not a valid partition address.

History:

Introduced in XtratuM 2.2.0.

See also:

`XM_get_partition_status` [Page: 18].

2.53 XM_trace_event

Synopsis: Records a trace entry.

Category:

Standard service.

Declaration:

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

Description:

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

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

A health monitoring log entry is a data `xmTraceEvent_t` structure.

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

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

Listing 2.11: /core/include/objects/trace.h

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

`moduleId`
: User defined field.

`criticality`
: Defines the importance of the trace event.

`code`
: An number which identifies cause of the trace.

The `event` shall be aligned to 8 bytes.

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

Return value:

`[XM_OK]`
: The trace event has been stored successfully.

[XM_INVALID_PARAM]
event address is not a valid.

History:

Introduced in XtratuM 2.2.0.

See also:

XM_trace_open [Page: 71], XM_trace_read [Page: 72], XM_trace_seek [Page: 73], XM_trace_status [Page: 74].

2.54 XM_trace_open

Synopsis: Open a trace stream.

Category:

Standard service.

Declaration:

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

Description:

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

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

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

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

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

Return value:

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

`[XM_PERM_ERROR]`
The calling partition is not supervisor.

`[XM_INVALID_PARAM]`
The `id` is not a valid partition address, or a valid trace stream name.

History:

Introduced in XtratuM 2.2.0.

See also:

`XM_trace_event` [Page: 69], `XM_trace_read` [Page: 72], `XM_trace_seek` [Page: 73], `XM_trace_status` [Page: 74].

2.55 XM_trace_read

Synopsis: Read a trace event.

Category:

Standard service.

Declaration:

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

Description:

Attempts to read one trace event from the stream `traceStream`. If no log entry is available, the call returns immediately.

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

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

A health monitoring log entry is a data `xmTraceEvent_t` structure.

Return value:

[0] There are no new log entries in the trace stream. No data is returned.

[1]
The next trace entry is returned and the current read position is increased by one.

[XM_PERM_ERROR]
The calling partition is not supervisor.

[XM_INVALID_PARAM]
The `traceEventPtr` address is not a valid partition address; or `traceStream` is not a valid trace stream descriptor.

History:

Introduced in XtratuM 2.2.0.

See also:

`XM_trace_event` [Page: 69], `XM_trace_open` [Page: 71], `XM_trace_seek` [Page: 73], `XM_trace_status` [Page: 74].

2.56 XM_trace_seek

Synopsis: Sets the read position in a trace stream.

Category:

Standard service.

Declaration:

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

Description:

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

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

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

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

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

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

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

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

Return value:

[XM_OK]
The operation succeeds.

[XM_PERM_ERROR]
The calling partition is not supervisor.

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

History:

Introduced in XtratuM 2.2.0.

See also:

`XM_trace_event` [Page: 69], `XM_trace_open` [Page: 71], `XM_trace_read` [Page: 72], `XM_trace_status` [Page: 74].

2.57 XM_trace_status

Synopsis: Get the status of a trace stream.

Category:

Standard service.

Declaration:

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

Description:

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

This service returns a structure which contains the following fields:

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

Listing 2.12: /core/include/objects/trace.h

Return value:

[XM_OK]
The operation succeeded.

[XM_PERM_ERROR]
The calling partition is not supervisor.

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

History:

Introduced in XtratuM 2.2.0.

See also:

`XM_trace_event` [Page: 69], `XM_trace_open` [Page: 71], `XM_trace_read` [Page: 72], `XM_trace_seek` [Page: 73].

2.58 XM_unmask_irq

Synopsis: Unmask interrupt.

Declaration:

```
xm_s32_t XM_unmask_irq(xm_u32_t noIrq)
```

Description:

This function unmasks (allows) the selected hardware or extended interrupt.

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

Return value:

[XM_OK]
The operation succeeded.

[XM_INVALID_PARAM]
If noIrq is invalid.

See also:

XM_mask_irq [Page: 42].

2.59 XM_update_page32

Synopsis: Writes `val` in `pAddr`.

Category:

Standard service.

Declaration:

```
xm_s32_t XM_update_page32(xmAddress_t pAddr, xm_u32_t val);
```

Description:

Pages declared as page table are marked as read-only, preventing the partition to freely update its content. Updating an entry in these pages requires the use of the hypercall `XM_update_page32`.

`pAddr` is the physical address to be updated, and `val` is the new value.

Before performing any change to the page, XtratuM validates that `val` does not breaks any security rule as mapping pages which do not belong to the partition, etc. If any rule is broken, the update is done successfully and the hypercall returns with `XM_OK`. Otherwise, `XM_INVALID_PARAM` is returned.

A partition can update the content of any page, no matter the type of the page by using this hypercall. The rules described above are only applied to the pages marked as page table ones.

Return value:

[`XM_OK`] The operation finished successfully.

[`XM_INVALID_PARAM`]The page `pAddr` does not belong to the partition or its content breaks some rule for its new `type`

Rationale:

Any partition has read-access to its own page tables avoiding the overhead of hypercall anytime a query to them. However, the page tables are marked as read-only, so updating any entry requires the partition to invoke `XM_update_page32`. This enable XtratuM to validate any update performed to the page tables.

Usage examples:

```
#include <xm_inc/arch/physmm.h/>
#include <xm.h>

unsigned long pgd[1024] __attribute__((aligned(4096)));
unsigned long pgt[1024] __attribute__((aligned(4096)));
...

void PartitionMain(void) {
   memset(ptdL1, 0, 4096);
   memset(ptdL2, 0, 4096);
   if (XM_set_page_type(ptdL1, PPAG_PTD1)!=XM_OK) {
      xprintf("ptdL1 couldn't be set as PTD1\n");
      XM_halt_partition(XM_PARTITION_SELF);
   }
   if (XM_set_page_type(ptdL2, PPAG_PTD2)!=XM_OK) {
      xprintf("ptdL2 couldn't be set as PTD2\n");
      XM_halt_partition(XM_PARTITION_SELF);
   }
```

```
    if (XM_update_page32(&ptdL1[0], (unsigned long)ptdL2|0x2)!=
        XM_OK) { // PTDL2 PRESENT
      xprintf("ptdL1[0] couldn't be updated\n");
      XM_halt_partition(XM_PARTITION_SELF);
    }
    if (XM_update_page32(&ptdL2[0], 0x40100000|0xe)!=XM_OK) { //
        PAGE PRESENT | USER_RW
      xprintf("ptdL2[0] couldn't be updated\n");
      XM_halt_partition(XM_PARTITION_SELF);
    }

     ...
}
```

See also:

XM_set_page_type [Page: 59]

2.60 XM_write_console

Synopsis: Print a string in the hypervisor console.

Category:

Standard service.

Declaration:

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

Description:

Writes the up to `length` bytes from the buffer pointed `str` to the default output console of XtratuM.

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

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

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

Return value:

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

Usage examples:

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

2.61 XM_write_object

Synopsis: Performs a write on a object.

Category:

Standard service.

Declaration:

```
xm_s32_t XM_write_object(xmObjDesc_t objDesc, void *buffer,
    xm_u32_t size, xm_u32_t *flags);
```

Description:

This hypercall writes `size` bytes starting from the address pointed by `buffer` to the the `objDesc` object.

Return value:

[XM_INVALID_PARAM]

- The `buffer` is out of range.
- The `size` is out of range.
- `objDesc` is not a valid object descriptor or the `objDesc` object has `write` disabled.

See also:

XM_read_object [Page: 49], XM_seek_object [Page: 57], XM_ctrl_object [Page: 13]

2.62 XM_write_register32

Synopsis: Modify a processor control register.

Category:

Standard service.

Declaration:

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

Description:

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

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

Below is the list of valid processor registers:

```
#define CR0_REG32 0
#define CR3_REG32 1
#define CR4_REG32 2
#define GDT_REG32 3
#define TSS_REG32 4
#define TLB_REG32 5
#define WBINVD_REG32 6
#define NR_REGS32 7
```

Listing 2.13: /core/include/ia32/processor.h

Return value:

[XM_OK]

The requested operation has been successful.

[XM_INVALID_PARAM]

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

2.63 XM_write_sampling_message

Synopsis: Writes a message in the specified sampling port.

Category:

Standard service.

Declaration:

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

Description:

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

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

Return value:

[XM_OK]
If the message has been successfully written into internal buffer.

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

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

Optimization:

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

Usage examples:

```
xm_s32_t portDesc, retCode;

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

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

See also:

`XM_create_sampling_port` [Page: 12], `XM_read_sampling_message` [Page: 50], `XM_get_sampling_port_status` [Page: 22], .