smx®

Reference Manual

Version 5.2.1

October 2024

by Ralph Moore

Micro Digital Associates, Inc.
(714) 437-7333
support@smxrtos.com
www.smxrtos.com

.

Table of Contents

smx Services 1
- Services Format 1
- Notes and Restrictions 2

smx_Block and smx_BlockPool 3
- smx_BlockGet 3
- smx_BlockMake 4
- smx_BlockPeek 5
- smx_BlockRel 6
- smx_BlockRelAll 7
- smx_BlockUnmake 7
- smx_BlockPoolCreate 9
- smx_BlockPoolDelete 10
- smx_BlockPoolPeek 10

smx_EVB 12
- smx_EVBInit 12
- smx_EVB_LOG Macros 12

smx_EventFlags and smx_EventGroup 14
- smx_EventFlagsPulse 14
- smx_EventFlagsSet 14
- smx_EventFlagsTest 16
- smx_EventFlagsTestStop 17
- smx_EventGroupClear 18
- smx_EventGroupCreate 19
- smx_EventGroupDelete 20
- smx_EventGroupPeek 20
- smx_EventGroupSet 21

smx_EventQueue 22
- smx_EventQueueClear 22
- smx_EventQueueCount 22
- smx_EventQueueCountStop 24
- smx_EventQueueCreate 26
- smx_EventQueueDelete 26
- smx_EventQueueSignal 27

smx_Heap 29
- smx_HeapBinPeek 29
- smx_HeapBinScan 30
- smx_HeapBinSeed 31
- smx_HeapBinSort 32
- smx_HeapCalloc 34
- smx_HeapChunkPeek 35
- smx_HeapExtend 36
- smx_HeapFree 37

smx_HeapInit 39
smx_HeapMalloc 42
smx_HeapPeek 44
smx_HeapRealloc 45
smx_HeapRecover 47
smx_HeapScan 48
smx_HeapSet 50
smx_HT 52
smx_HT 52
smx_ISR 54
smx_ISR_ENTER 54
smx_ISR_EXIT 56
smx_LSR 57
smx_LSRCreate 57
smx_LSRDelete 58
smx_LSRInvoke 58
smx_LSRsOff 60
smx_LSRsOn 60
smx_Msg 61
smx_MsgBump 61
smx_MsgGet 62
smx_MsgMake 63
smx_MsgPeek 64
smx_MsgReceive 65
smx_MsgReceiveStop 66
smx_MsgRel 68
smx_MsgRelAll 69
smx_MsgSend 70
smx_MsgUnmake 72
smx_MsgXchg 74
smx_MsgXchgClear 74
smx_MsgXchgCreate 75
smx_MsgXchgDelete 76
smx_MsgXchgPeek 76
smx_MsgXchgSet 77
smx_Mutex 79
smx_MutexClear 79
smx_MutexCreate 80
smx_MutexDelete 80
smx_MutexFree 81
smx_MutexGet 82
smx_MutexGetFast 83
smx_MutexGetStop 84
smx_MutexRel 85
smx_MutexRelFast 85
smx_Pipe 87
smx_PipeClear 87
smx_PipeCreate 87
smx_PipeDelete 88
smx_PipeGet8 89
smx_PipeGet8M 90
smx_PipeGetPkt 91

smx_PipeGetPktWait 92
smx_PipeGetPktWaitStop 94
smx_PipePeek 96
smx_PipePut8 97
smx_PipePut8M 98
smx_PipePutPkt 99
smx_PipePutPktWait 100
smx_PipePutPktWaitStop 101
smx_PipeResume 103
smx_PipeSet 104
smx_Sem 106
smx_SemClear 106
smx_SemCreate 106
smx_SemDelete 107
smx_SemPeek 108
smx_SemSet 109
smx_SemSignal 110
smx_SemTest 111
smx_SemTestStop 114
smx_SSR 116
smx_SSR_ENTER 116
smx_SSR_EXIT 117
smx_Sys 118
smx_SysEtimeGet 118
smx_SysPseudoHandleCreate 118
smx_SysPeek 119
smx_SysStimeGet 119
smx_SysPowerDown 119
smx_SysWhatIs 121
smx_Task 122
smx_TaskBump 122
smx_TaskCreate 123
smx_TaskCurrent 125
smx_TaskDelete 126
smx_TaskLocate 127
smx_TaskLock 128
smx_TaskLockClear 129
smx_TaskPeek 130
smx_TaskResume 131
smx_TaskSet 132
smx_TaskSleep 133
smx_TaskSleepStop 134
smx_TaskStart 135
smx_TaskStop 138
Task or LSR Autostop 139
smx_TaskSuspend 140
smx_TaskUnlock 141
smx_TaskUnlockQuick 142
smx_TaskYield 143
smx_Timer 144
smx_TimerDup 144
smx_TimerPeek 145
smx_TimerReset 146

smx_TimerSetLSR 147
smx_TimerSetPulse 148
smx_TimerStart 149
smx_TimerStop 151
smx Utility Functions 152
smx_ConvMsecToTicks 152
smx_ConvTicksToMsec 152
smx_DelayMsec 152
smx_DelayTicks 153
smx_ERROR 153
smx_EBDisplay 154
smx_Go 154
smx Glossary 155

smx Services

This section covers all smx system services, including SSRs, functions, and macros. For simplicity, these are often referred to as *smx calls* or just *calls*. Each description provides all information necessary to properly use the subject smx call. Read the smx User's Guide for information concerning the theory and application of smx services. The smx Glossary at the end of this manual defines all smx terms and symbols.

Names in all caps are generally data types, manifest constants, macros, or enumerated constants. Names such as *atask* are handles for objects such as tasks. Hence, atask->afield can be used to access *afield* in atask control block. It is also possible to access afield via the control block structure: tcb.afield. However, it is better to use the Peek functions provided by smx to access object information, since direct field accesses cannot be used in SecureSMX systems. A function is identified by parentheses after the name — for example smx_TaskStart().

Services Format

The synopsis of the call is listed first. It employs the ANSI standard for function prototypes. Following it are these fields:

Type	Indicates whether the call is an SSR, macro, function, etc. See the *smx Glossary* section for discussion of call types.
Summary	Summary of what the call does.
Compl	Complementary call. This is the call that performs the inverse operation, if any.
Parameters	Describes the parameters of the call, if any.
Returns	Shows what, if anything, is directly returned by the call. If 0, FALSE, or NULL is returned, it may be assumed that the call has been aborted and that nothing has been changed, unless otherwise indicated.
Errors	Lists the error types which may occur for the call. See the Glossary for descriptions of error types. If an error is detected the service is aborted and FALSE or NULL is returned, unless otherwise indicated. Some secondary errors (from called SSRs or smxBase functions) may not be listed, but can occur during operation.
Descr	Description of the call. It helps when reading a call description to remember that if a call is made from a task, that task is the current task (smx_ct) while the call is executing. Similarly, if a call is made from an LSR, that LSR is the current LSR (smx_clsr) while the call is executing.
Notes	Specifics concerning control block fields, etc. This information is not necessary to properly use the call, but may be helpful for debugging or for better understanding.
TaskMain	The prototype for the task's main function (for the task being stopped) is shown here for a stop call, as well as the parameter passed to the task's main function when the task is restarted. The parameter to the main function can be void if you do not need to reference the value passed in. This is the case when specifying an infinite timeout (SMX_TMO_INF) in the stop call. Otherwise, you are advised to do error checking and handle the case where the stop call times out.
Example	These are intended to illustrate the common uses of calls. As such, they are often unencumbered with error checking. See the Error Management chapter of the smx User's Guide for discussion of error checking.

Notes and Restrictions

(1) Timeouts are specified in ticks. They may be specified in milliseconds by ORing with flag SMX_FL_MSEC, such as 10|SMX_FL_MSEC. The granularity is ticks, which is rounded up if necessary (e.g. 15 msec = 2 ticks for 100 ticks/sec.)

(2) Any code following a stop call will not execute. When the task is restarted, execution starts at the beginning of the task's main function with the return value from the call passed in as the task main parameter. Note: stop and start calls which specify the task are an exception to this.

(3) Stop SSRs: The parameter of the task main function should be the same type as the return value of the suspend SSR — e.g. void task_main(MCB_PTR msg).

(4) Bare functions should not be used in tasks because they are not protected from preemption.

(5) Bare functions and SSRs may not be mixed on the same end of a pipe (e.g. having an ISR and a task both putting packets into the same pipe).

(6) Task and LSR main function parameters: On some processors, such as ColdFire, there are separate address and data registers. If the compiler passes parameters in registers rather than on the stack, you must define the parameter to be a data type (e.g. integer) rather than a pointer. (Note that smx handles are pointers.) If it is necessary to pass a pointer, define the type to be u32 and then typecast it to the pointer type within the task or LSR main function, as follows:

```
void  task_main(u32 par)
{
      MCB_PTR msg = (MCB_PTR)par;
      /* use msg */
}
```

(7) The handle pointer (hp) parameter of SSRs that create or get objects is used to prevent multiple creates of objects. Handles must be NULL or smx_nullcb when calling Create or Get SSRs, in order to avoid aborting the SSR and generating a SMXE_INV_OP error. hp can also used by Create or Get SSRs to directly load handles instead of loading the SSR return value. Under SecureSMX hp is used to check that the caller has a token that permits changing or accessing an smx object. Autovariable handles must be explicitly cleared before using in a Create or Get call to avoid this error.

smx_Block and smx_BlockPool

See the smxBase User's Guide for Base Block Pool functions, and see the smx User's Guide, Memory Management chapter for usage information and more examples.

smx_BlockGet

BCB_PTR smx_BlockGet (PCB_PTR pool, u8** bpp, u32 clrsz=0, BCB_PTR* bhp=NULL)

Type SSR

Summary Gets an smx block by combining a data block from a block pool and a BCB from the BCB pool.

Compl smx_BlockRel()

Parameters

pool	Pool to get block from.
bpp	Pointer to block pointer. NULL if none.
clrsz	Number of bytes to clear from the start of block.
bhp	Block handle pointer (see hp note in Notes and Restrictions).

Returns

blk	Handle of smx block obtained.
NULL	No block available or error.

Errors

SMXE_INV_PCB	Invalid pool handle.
SMXE_OUT_OF_BCBS	
SMXE_INV_OP	Attempted multiple gets of same block.

Descr Gets a block from the specified block pool for use as the data block and a BCB from the BCB pool, initializes the BCB and links it to the data block. Clears the first clrsz bytes of the data block up to its size and loads the address of the data block into bpp, unless it is NULL. bpp can be used to load data into the data block. The current task or LSR becomes the smx block owner. Returns the block handle.

Notes

1. For proper operation there must be at least as many BCBs as there are active smx blocks in a system at any given time.
2. Interrupt safe with respect to sb_BlockGet() and sb_BlockRel() operating on the same block pool.

Example

```
BCB_PTR build_msg(PCB_PTR pool)
{
    u8*         dbp;
    BCB_PTR     blk;
    blk = smx_BlockGet(pool, &dbp, 4);
    /* load blk using dbp */
    return blk;
}
```

This function gets a message from pool, loads data into it, and returns the block handle.

smx_BlockMake

BCB_PTR smx_BlockMake (PCB_PTR pool, u8* bp, BCB_PTR* bhp=NULL)

Type SSR

Summary Makes an smx block from a base block or a bare block using a pointer to it.

Compl smx_BlockUnmake()

Parameters

pool	Base pool of the block. NULL if none.
bp	Block pointer.
bhp	Block handle pointer (see hp note in Notes and Restrictions).

Returns

blk	Handle of block obtained.
NULL	Insufficient resources or error.

Errors

SMXE_OUT_OF_BCBS	
SMXE_INV_OP	Attempted multiple makes or creates of same block.
SMXE_INV_PAR	bp is NULL or out of pool range, if pool not NULL.

Descr Makes a block from a bare block, using its pointer. Gets BCB from BCB pool, initializes it and returns its handle. The pool pointer or NULL, if no pool, is stored in the BCB.

Notes

1. The pool parameter is not used in this operation. It can be supplied so that a base block can be released back to the correct pool, at a later time.
2. For proper operation there must be at least as many BCBs as there are active blocks in a system at any given time.
3. Bare blocks can be statically defined, obtained from a base block pool, DAR, heap, or ROM, or any other source.

Example:

```
#define WIDTH 4;
#define LENGTH 20;

LCB_PTR     in_LSR;
PCB         in_pool;        /* base pool */
PICB_PTR    in_pipe;
u8 pp[WIDTH*LENGTH];

in_LSR = smx_LSRCreate(in_LSR_main, NULL, 0, SMX_FL_TRUST, “in_LSR”);
in_pipe = smx_PipeCreate(&pp, WIDTH, LENGTH, "in_pipe");

void in_ISR(void)
{
    static u8 *bp, *dp;
    u8 ch = UART_In();

    switch (ch)
    {
        case: STX
            bp = sb_BlockGet(&in_pool, 4);
            dp = bp;
```

```
                break;
            case: ETX
                smx_LSR_INVOKE(in_LSR, (u32)bp)
                break;
            default:
                *dp++ = ch;
        }
    }

    void in_LSR_main(u32 bp);
    {
        BCB_PTR blk;

        blk = smx_BlockMake(&in_pool, (u8*)bp);
        if (!smx_PipePutPktWait(in_pipe, &blk, NO_WAIT))
        smx_BlockRel(blk, 0);
    }
```

in_ISR() runs whenever a UART input interrupt occurs. It gets the incoming character from the UART. If it is the start of text (STX) a base block is obtained from in_pool. Subsequent characters are loaded into the base block. When the end of text (ETX) is received, in_LSR is invoked. in_LSR uses smx_BlockMake() to make the base block at bp into an smx block and then puts its handle, blk, into in_pipe where a task waits to process it. Note that this is a no-copy operation. Note also, that if in_pipe is full, the block is released so a memory leak will not occur. Unfortunately, the data is also lost.

smx_BlockPeek

u32 smx_BlockPeek (BCB_PTR blk, SMX_PK_PAR par)

Type SSR

Summary Returns the current value of the parameter specified.

Parameters

blk	Block to peek at.
par	What to return.

Returns

value	Value of par.
0	Value, unless error.

Errors

SMXE_INV_BCB	Invalid block handle
SMXE_INV_PAR	Invalid argument.

Notes This service can be used to peek at a block. Valid arguments are:

SMX_PK_BP	Block pointer.
SMX_PK_NEXT	Next block in the free list, if block is free, else 0.
SMX_PK_ONR	Block owner, 0 if none.
SMX_PK_POOL	Block pool, 0 if note.
SMX_PK_SIZE	Block size.

Example

```
TCB_PTR task;

task = (TCB_PTR)smx_BlockPeek(blk, SMX_PK_ONR);
if (task == smx_ct)
    smx_BlockRel(blk, 0);
```

smx_BlockRel

BOOLEAN smx_BlockRel (BCB_PTR blk, u16 clrsz=0)

Type SSR

Summary Releases a block obtained by smx_BlockGet() or made by smx_BlockMake().

Compl smx_BlockGet(), smx_BlockMake()

Parameters

	blk	Block to release.
	clrsz	Number of bytes to clear from byte 4 of block.

Returns

TRUE	Block released.
FALSE	Block not released due to error.

Errors

SMXE_INV_BCB	blk is invalid or block has already been released.
SB_INV_PCB	Invalid pool handle.
SB_INV_BP	blk->bp is not in pool.

Descr Releases a block obtained by smx_BlockGet() or made by smx_BlockMake(). Releases the data block back to its base pool, if pool is valid. In this case, the blk->ph must point to a pool control block. Clears clrsz bytes from byte 4 to the end of the block. Also releases the BCB back to its pool. If blk->bhp is non-zero, smx_nullcb is loaded into *blk->bhp.

Notes

1. This SSR can be used to release an smx block, which was made from a bare block.
2. Loading smx_nullcb into *blk->bhp prevents accidental reuse of blk and shows clearly that blk has been released.
3. Interrupt safe with respect to sb_BlockGet() and sb_BlockRel() operating on the same base block pool.

Example

```
BCB_PTR blk;
u32 sz;

sz = smx_BlockPeek(blk, SMX_PK_SIZE);
smx_BlockRel(blk, sz);
```

This clears the data block of blk, except the first 4 bytes, and releases it. (Note: The first 4 bytes of a free data block are used for the free list link to the next block.)

smx_BlockRelAll

u32 smx_BlockRelAll (TCB_PTR task)

Type SSR

Summary Releases all blocks owned by task and returns the number released.

Parameters

task	Task whose blocks are to be released.

Returns

n	Number of blocks released.
0	Error or no blocks were owned

Errors SMXE_INV_TCB Invalid task handle.

Descr Searches the BCB pool and releases all blocks owned by task. Returns the number of blocks released.

Example

```
void stop_task(TCB_PTR atask)
{
    smx_BlockRelAll(atask);
    smx_TaskStop(atask);
}
```

smx_TaskStop(atask) does not automatically release all blocks owned by atask. In this example, all of atask's blocks are released, then it is stopped. This prevents block leakage if the task gets the blocks again when it is restarted.

smx_BlockUnmake

u8* smx_BlockUnmake (PCB_PTR* pool, BCB_PTR blk)

Type SSR

Summary Unmakes a block made by smx_BlockMake() into a bare block.

Compl smx_BlockMake(), smx_BlockGet()

Parameters

pool	Pointer to pool handle, if any.
blk	Block to unmake.

Returns

>0	Block unmade.
NULL	Error.

Errors SMXE_INV_BCB Invalid block handle or block already unmade or released.

Descr Unmakes an smx block made by smx_BlockMake() or a block obtained by smx_BlockGet() by converting it into a bare block and releasing its BCB. If pool != NULL, loads blk->ph into *pool, so the code receiving a base block can get its handle. If pool == NULL, the block is a bare block. If blk->bhp != NULL, loads smx_nullcb into *blk->bhp.

Notes

1. Interrupt safe with respect to sb_BlockGet() and sb_BlockRel() operating on the same block pool.

2. Loading smx_nullcb into *blk->bhp prevents accidental reuse of blk and shows clearly that blk has been released.

Example

```
LCB_PTR     out_LSR;
PICB_PTR    out_pipe;
u8*         pkt_ptr;
PCB_PTR     pkt_pool;
u32         pkt_sz;
u32         bp;

out_LSR = smx_LSRCreate(out_LSR_main, NULL, 0, SMX_FL_TRUST, “out_LSR”);
smx_LSRInvoke(out_LSR, (u32)out_pipe);

void out_LSR_main(u32 pipe);
{
    BCB_PTR pkt;

    if (smx_PipeGetPkt((PICB_PTR)pipe, (u8*)&pkt))
    {
        pkt_sz = smx_BlockPeek(pkt, SMX_PK_SIZE);
        pkt_ptr = smx_BlockUnmake(&pkt_pool, pkt);
        bp = pkt_ptr;
        pkt_sz--;
        UART_Out(bp++);
    }
}

void out_ISR(void);
{
    if (pkt_sz > 0)
    {
        pkt_sz--;
        UART_Out(bp++);
    }
    else
    {
        UART_Stop();
        sb_BlockRel(pkt_pool, pkt_ptr, 0);
    }
}
```

This example is the opposite of that shown for smx_BlockMake(). It is assumed that a task invokes out_LSR when it puts a packet handle into out_pipe. out_LSR gets the next packet handle from out_pipe and puts it into pkt. It then determines pkt_sz and unmakes pkt into a bare block at pkt_ptr and puts the pool handle into pkt_pool. out_LSR decrements pkt_sz and outputs the first byte to the UART to start the UART send.

The UART interrupts each time it needs another byte, and out_ISR provides the next byte until all bytes have been sent. out_ISR then stops the UART and releases the bare block back to pkt_pool. pkt_pool could be an smx block pool, a base block pool, or NULL. In the latter case, the block is not released to any pool.

smx_BlockPoolCreate

PCB_PTR smx_BlockPoolCreate (u8* pp, u8 num, u16 size, const char* name=NULL, PCB_PTR* php=NULL)

Type SSR

Summary Creates an smx block pool of num size blocks at pp.

Compl smx_BlockPoolDelete()

Parameters

pp	Pointer to memory for pool.
num	Number of blocks.
size	Size of blocks.
name	Name to give block pool, NULL for none.
php	Pool handle pointer (see hp note in Notes and Restrictions).

Returns

pool	Pool handle.
NULL	Insufficient resources or error.

Errors

SMXE_INV_OP	Attempted multiple creates of the same block pool.
SMXE_INV_PAR	Invalid parameter.
SMXE_OUT_OF_PCBS	

Descr Gets PCB for pool and calls sb_BlockPoolCreate() to creates block pool of num*size blocks at pp and loads name. If successful, initializes PCB and returns block pool handle. If not, returns PCB to its pool.

Notes

1. pp must be 4-byte aligned.
2. The block pool can be created from any block anywhere in memory, and it is assumed to be large enough.

Example

```
#define NUM 100;
#define SIZE 20;

PCB_PTR poolA;

u8 p = &pa[NUM*SIZE];   /* static pool */
  -or-
u8* p = (u8*)smx_HeapCalloc(NUM, SIZE);   /* heap pool */

poolA = smx_BlockPoolCreate(p, NUM, SIZE, "poolA", &poolA);
```

Creates a block pool of NUM blocks, of SIZE bytes in either a static block of memory or in a block allocated from the heap. Note that if the smx_HeapCalloc fails, p == 0 and smx_BlockPoolCreate() will also fail.

smx_BlockPoolDelete

u8* smx_BlockPoolDelete (PCB_PTR* php)

Type SSR

Summary Deletes an smx block pool.

Compl smx_BlockPoolCreate()

Parameters

php	Pool handle pointer.	

Returns

>0	Pool deleted.
NULL	Error.

Errors

SMXE_BLK_IN_USE	One or more blocks are still in use.
SMXE_INV_PCB	Invalid block pool handle.

Descr Deletes a block pool created by smx_BlockPoolCreate(). Clears and releases its PCB, sets *php = smx_nullcb so it cannot be used again, and returns a pointer to the start of the released pool.

Note User is responsible for dealing with the pool block.

Example

```
u8*         bp;
PCB_PTR     poolA;

bp = smx_BlockPoolDelete(&poolA);
smx_HeapFree((void*)bp);
```

If the pool delete fails, bp will be NULL and smx_HeapFree() will do nothing. If bp is not within the heap, smx_HeapFree() will abort with an error.

smx_BlockPoolPeek

u32 smx_BlockPoolPeek (PCB_PTR pool, SMX_PK_PAR par)

Type SSR

Summary Returns the current value of the parameter specified.

Parameters

pool	Block pool to peek at.
par	What to return.

Returns

value	Value of par.
0	Value, unless error.

Errors

SMXE_INV_PCB	Invalid block pool handle.

SMXE_INV_PAR Invalid parameter.

Notes This service can be used to peek at a block pool. Valid arguments are:

SMX_PK_NUM	Number of blocks in pool.
SMX_PK_NUM_USED	Number of used blocks in pool.
SMX_PK_FREE	Number of free blocks in pool.
SMX_PK_FIRST	First free block in pool.
SMX_PK_MIN	First physical block in pool.
SMX_PK_MAX	Last physical block in pool.
SMX_PK_NAME	Name of the pool.
SMX_PK_SIZE	Size of the blocks in pool.

Example

```
SCB_PTR semA;
void app_init(void)
{
    u32 lim = smx_BlockPoolPeek(poolA, SMX_PK_NUM);
    semA = smx_SemCreate(SMX_SEM_RSRC, lim, "sr");
}
```

This shows using smx_BlockPoolPeek() during initialization of semA, which is used to control access to poolA.

smx_EVB

See the smx User's Guide, Event Logging chapter for usage information and more examples.

smx_EVBInit

void smx_EVBInit (u32 flags)

Type Bare function

Summary Creates and initializes the Event Buffer, EVB.

Parameters flags Flags indicate what to log:

SMX_EVB_EN_TASK	
SMX_EVB_EN_LSR	
SMX_EVB_EN_ISR	
SMX_EVB_EN_ERR	
SMX_EVB_EN_USER	
SMX_EVB_EN_PORTAL	
SMX_EVB_EN_PERR	
SMX_EVB_EN_SSR1-8	SSR groups 1-8.
SMX_EVB_EN_SSRS	All SSR groups.
SMX_EVB_EN_ALL.	

Returns none

Descr Initializes the event buffer, EVB, and specifies which types of events to log. Space for EVB is allocated by the linker command file. The EVB_EN flags can also be changed via smxAware. The PORTAL and PERR enables are for usage with SecureSMX.

Example

```
smx_EVBInit(SMX_EVB_EN_ALL);   /* enable logging of all events */

smx_EVBInit(SMX_EVB_EN_ERR +
            SMX_EVB_EN_ISR +
            SMX_EVB_EN_LSR);   /* enable logging of errors, ISRs, and LSRs */
```

smx_EVB_LOG Macros

void smx_EVB_LOG_ISR (u32 isr)
void smx_EVB_LOG_ISR_RET (u32 isr)
void smx_EVB_LOG_LSR (u32 handle)
void smx_EVB_LOG_LSR_RET (u32 handle)
void smx_EVB_LOG_SSRn (u32 id, u32 p1, …, pn)
void smx_EVB_LOG_USERn (u32 handle, u32 p1, …, pn)

Type Bare macros that call functions

Summary Record respective events in the Event Buffer.

Parameters

handle	Task or LSR handle or user pseudo handle.
id	SSR ID
isr	ISR pseudo handle.
p1-n	SSR parameter or user value.

Returns None

Descr These macros load events into EVB. The ISR macros log ISR pseudo handles. The macros must be put into the ISRs in order to log them and the ISR enable must be set. The LSR macros log the LSR handle and task macros log the task handle. These are called automatically, if their enables have been set.

The user macros can be used anywhere in the code to serve as timestamps and to show values of up to n variables. The handle of a user event can be a pseudo handle or some other unique identifier, chosen by the user.

The ISR and LSR macros are written as macros, for speed; the others call functions, in order to minimize code space. See xevb.h and xevb.c. For ISRs written using an assembly shell, put the above C macros in the C body of the ISR. Specifically, put smx_EVB_LOG_ISR() right after smx_ISR_ENTER() and smx_EVB_LOG_ISR_RET() right before smx_ISR_EXIT(). These will mark the time spent in the body of the ISR.

Example

```
void appl_init(void)
{
    void* isr1_h = smx_SysPseudoHandleCreate();
    smx_HT_ADD(isr1_h, "isr1");
}

void isr1(void
{
    smx_EVB_LOG_ISR(isr1);
    /* isr1 code */
    smx_EVB_LOG_LSR_RET(isr1);
}
```

The above example shows how to log a non-smx ISR into the Event Buffer. A pseudo handle is created for it because ISRs do not have handles.

smx_EventFlags and smx_EventGroup

See the smx User's Guide, Event Groups chapter for usage information and more examples.

smx_EventFlagsPulse

BOOLEAN smx_EventFlagsPulse (EGCB_PTR eg, u32 pulse_mask)

Type SSR

Summary Pulses event flags on and off that are not already set.

Compl smx_EventFlagsTest() and smx_EventFlagsTestStop()

Parameters

eg	Event flags group.
pulse_mask	Flags to pulse.

Returns

TRUE	Flags pulsed.
FALSE	Flags not pulsed.

Errors SMXE_INV_EGCB Invalid event group handle.

Descr See smx_EventFlagsSet() for operational description. This service is useful in situations where it is desired to resume tasks that are already waiting for a specific combination (AND, OR, or ANDOR) of the specified flags. It does not have a pre-clear mask. If a pulsed flag was already set, it will be left set unless cleared by a post clear flag of a resumed task. If a pulsed flag was reset, it will be left reset.

Example

```
#define F2      0x2
#define F1      0x1
EGCB    eg;

void t2aMain(u32)
{
      smx_EventFlagsPulse(eg, F2+F1);
}
```

Resumes tasks already waiting for a combination of F2 and F1. If a task was waiting for either or both, it will be resumed.

smx_EventFlagsSet

BOOLEAN smx_EventFlagsSet (EGCB_PTR eg, u32 set_mask, u32 pre_clear_mask)

Type SSR

Summary Clears flags in eg selected by 1 bits in pre_clear_mask, sets flags selected by 1 bits in set_mask, and resumes waiting tasks which now match a pre-specified combination of eg->flags.

Compl smx_EventFlagsTest() and smx_EventFlagsTestStop()

Parameters

eg	Event group.
set_mask	Flags to set.
pre_clear_mask	Flags to pre-clear

Returns

TRUE	Flags cleared and set.
FALSE	Flags not cleared and set.

Errors SMXE_INV_EGCB Invalid event group handle.

Descr Pre-clears flags selected by pre_clear_mask in eg, and sets flags selected by set_mask. Then, if at least one new flag has been set, the task wait queue is searched for matches to eg->flags. Each task's test_mask, post_clear_mask, and test condition (OR, AND, or ANDOR) are obtained from its TCB. The test mask condition is compared to eg->flags and if there is a match, the task is resumed. The flags causing the match are recorded in the rv field of the TCB and will be returned when the task starts running. (They are the return value of the test operation, which caused the task to wait.)

After this, the match flags are ANDed with the post_clear_mask for the task. The result of the AND is the *reset mask* for the task. For example: if flags causing a match = M & A and the post_clear_mask = A, then the result is A. This allows auto-clearing event flags, like A, without auto-clearing mode flags, like M.

If there are multiple tasks waiting, the above procedure is repeated for each task. When all tasks have been processed, their reset masks are ORed; then the 1's complement of the OR is ANDed with eg->flags. Thus all flags causing matches, after AND'ing with corresponding post_clear_mask, are reset. See smx_EventFlagsTest() for more discussion and examples.

If eg->cbfun is not NULL, the callback function cbfun(EGCB_PTR eg) is called. See also smx_EventGroupSet().

Example

```
#define TXRDY 0x40

EGCB modem_eg;

void start_transmit(void)
{
    smx_EventFlagsSet(modem_eg, TXRDY, 0);
}
```

Sets transmit ready flag in the modem_eg event group and resumes any tasks waiting for it. There is no pre-clear, in this case.

smx_EventFlagsTest

u32 smx_EventFlagsTest (EGCB_PTR eg, u32 test_mask, u32 mode, u32 post_clear_mask, u32 timeout=0)

Type SSR

Summary Tests for a match between eg->flags and test_mask. If found, smx_ct is continued. Returns the flags causing the match, and clears flags causing a match that are selected by the post clear_mask. Suspends smx_ct if no match is found and timeout > 0.

Compl smx_EventFlagsSet() and smx_EventFlagsPulse().

Parameters

eg	Event group.
test_mask	Flags to test.
mode	SMX_EF_OR, SMX_EF_AND, or SMX_EF_ANDOR.
post_clear_mask	Flags to reset of those causing a match.
timeout	Timeout in ticks or msec if |SMX_FL_MSEC.

Returns

flags	Flags causing match.
0	No match, timeout, or error.

Errors

SMXE_INV_EGCB	Invalid event group handle.
SMXE_INV_PAR	test_mask == 0.
SMXE_WAIT_NOT_ALLOWED	Call from LSR with timeout > 0.

Descr Tests flags in event group vs. test_mask. If match, clears matching flags selected by post-clear_mask, continues task, and returns flags which caused the match. If test_mask bit 16 is 1 (0x10000), tests for the AND of flags. mode determines the test to make. Bits 31 - 0 are the test pattern to compare to eg->flags. For AND/OR testing, flags in AND terms must be adjacent. For example, ABC, AB + C, or A +BC can be tested for, but not AC + B. For efficiency, terms should be as close to the least significant end, as possible.

If called from smx_ct and there is no match and timeout > 0, saves test_mask in ct->sv and the post_clr_mask in sv2. If AND, sets ct->flags.ef_and; if ANDOR, sets ct->flags.ef_andor; Enqueues task in FIFO order in eg wait queue, loads smx_ct timeout, and suspends smx_ct. If there is no match and no timeout, fails and returns 0.

If the timeout elapses the task resumes with 0 return value. Otherwise, when a match occurs, due to smx_EventFlagsSet() or smx_EventFlagsPulse() from another task or LSR, this task resumes with its return value equal to the flags that caused the match. It clears those flags that caused the match that are selected by the post_clear_mask

Operation from an LSR is the same as from a task except that waits are not allowed. Hence, an LSR can determine if flags are currently set, but it cannot wait for them.

Notes 1. Clears smx_lockctr if called from a task and timeout != SMX_TMO_NOWAIT.

Example

```
#define AND     SMX_EF_AND
#define TXRDY  0x4
#define DSR     0x2
#define CTS     0x1
#define TFLGS   0x7
u32  flags;
EGCB_PTR      modeg;

void TransmitMain(u32)
{
      while (flags = smx_EventFlagsTest(modeg, TFLGS, AND, TFLGS, 100))
      {
            if (flags == TFLGS)
                  /* send next message */
            else
                  break;  /* timeout -- stop sending */
      }
}
```

The transmit task waits for the modem flags: TXRDY, DSR, and CTS to all be TRUE. It then resets the flags, sends the next message, and waits upon them again. It stops transmitting if a timeout occurs.

smx_EventFlagsTestStop

void smx_EventFlagsTestStop (EGCB_PTR eg, u32 test_mask, u32 mode, u32 post_clear_mask, u32 timeout=0)

Type Limited SSR — tasks only.

Summary Same as smx_EventFlagsTest() except that smx_ct is always stopped, then restarted when it is time for it to run.

Compl smx_EventFlagsSet() and smx_EventFlagsPulse().

Parameters

eg	Event group.
test_mask	Flags to test.
mode	SMX_EF_OR, SMX_EF_AND, or SMX_EF_ANDOR.
post_clear_mask	Flags to reset of those causing a match.
timeout	Timeout in ticks or msec if |SMX_FL_MSEC.

Errors

SMXE_OP_NOT_ALLOWED	Called from an LSR
SMXE_INV_EG	Invalid event group handle.
SMXE_INV_PAR	test_mask == 0.

Descr See smx_EventFlagsTest() for operational description. ct always stops, then restarts instead of resuming. The flags causing a match are returned via the parameter in taskMain(par), when task restarts.

Notes 1. If called from an LSR, aborts operation and returns to the LSR.

2. Clears smx_lockctr if called from a task, since it always stops.

TaskMain void task_main(u32 par)

par flags Flags causing match.
0 No match.

Example The equivalent example of the above smx_EventFlagsTest() example is:

```
void TransmitMain(u32 par)
{
    if (par == TFLGS)
        /* send next message */
    else if (par == 0)
        smx_TaskStop(smx_ct);  /* timeout -- stop sending */
    smx_EventFlagsTestStop(modeg, TFLGS, AND, TFLGS, 100);
}
```

This task would initially be started as follows:

```
TCB_PTR transmit;
smx_TaskStart(transmit, 1);
```

The first time transmit runs it does not send a message, nor do a timeout stop. Instead, it tests the modem flags and stops. When there is a match, transmit will restart, and the matching flags will be passed into TransmitMain() as par. If a timeout or error occurs, transmit will stop since par == 0.

smx_EventGroupClear

BOOLEAN smx_EventGroupClear (EGCB_PTR eg, u32 init_mask)

Type SSR

Summary Clears event group.

Parameters eg Event group to clear.
init_mask Values to set flags.

Returns TRUE eg cleared.
FALSE eg not cleared.

Errors SMXE_INV_EG Invalid event group handle.

Descr Resumes all waiting tasks with 0 return values and sets eg->flags = init_mask. Typically used for system recovery.

Example

```
EGCB_PTR eg;
smx_EventGroupClear(eg, flags);
```

Clears eg task wait list and leaves eg->flags = flags.

smx_EventGroupCreate

EGCB_PTR smx_EventGroupCreate (u32 init_mask, const char* name=NULL, EGCB_PTR* eghp=NULL)

Type SSR

Summary Creates an event group with 32 flags.

Compl smx_EventGroupDelete()

Parameters

init_mask	Initial values of flags.
name	Name to give event group (NULL for none).
eghp	Event group handle pointer (see hp note in Notes and Restrictions).

Returns

handle	Event group created.
NULL	Event group not created.

Errors SMXE_OUT_OF_EGCBS

Descr Gets an event group control block from the EGCB pool and initializes it. Loads init_mask into EGCB flags field, name into EGCB name field, eghp into EGCB eghp field, and sets *eghp = eg. If allocation fails, returns NULL and reports SMXE_OUT_OF_EGCBS.

Example

```
#define CM 8
EGCB_PTR comm_eg;

void comm_init(void)
{
    comm_eg = smx_EventGroupCreate(CM, "comm_eg");
}
```

Creates an event group with handle and name comm_eg and flag CM set.

smx_EventGroupDelete

BOOLEAN smx_EventGroupDelete (EGCB_PTR* eghp)

Type SSR

Summary Deletes an event group.

Compl smx_EventGroupCreate()

Parameters

eghp	Event group handle pointer.

Returns

TRUE	Event group deleted.
FALSE	Event group not deleted.

Errors

SMXE_INV_EGCB	Invalid event group handle

Descr Deletes an event group created by smx_EventGroupCreate(). First resumes waiting tasks, giving them 0 return values and clearing their timeouts. Then clears the EGCB, returns it to the EGCB pool, and sets *eghp = smx_nullcb so it cannot be used again.

Example

```
EGCB_PTR eg;
smx_EventGroupDelete(&eg);
```

smx_EventGroupPeek

u32 smx_EventGroupPeek (EGCB_PTR eg, SMX_PK_PAR par)

Type SSR

Summary Returns the current value of the parameter specified.

Parameters

eg	Event group to peek.
par	What to return.

Returns

value	Value for par.
0	Value, unless error.

Errors

SMXE_INV_EGCB	Invalid event group handle.
SMXE_INV_PAR	Invalid parameter.

Notes This service can be used to peek at an event group. Valid arguments are:

SMX_PK_FLAGS	Flags
SMX_PK_TASK	Number of tasks waiting
SMX_PK_FIRST	First task waiting
SMX_PK_NAME	Name of event group

Example

```
EGCB_PTR    eg;
u32         num_tasks;
TCB_PTR     first_task;

num_tasks = smx_EventGroupPeek(eg, SMX_PK_TASK);
if (num_tasks > 0)
    first_task = (TCB_PTR)smx_EventGroupPeek(eg, SMX_PK_FIRST);
else
    first_task = smx_nullcb;
```

smx_EventGroupSet

BOOLEAN smx_EventGroupSet(EGCB_PTR eg, SMX_ST_PAR par, u32 v1, u32 v2)

Type SSR

Summary Provides event group control.

Compl smx_EventGroupPeek()

Parameters

eg	Event Group to set.
par	Parameter to set.
vl	Value 1.
v2	Value 2.

Returns

TRUE	Parameter has been set.
FALSE	Parameter has not been set due to error.

Errors

SMXE_INV_EGCB	Invalid event group handle.
SMXE_PRIV_VIOL	Privilege violation; cannot call from umode (SecureSMX).

Descr par is of type SMX_ST_PAR. Available parameters are:

SMX_ST_CBFUN	Event flags set callback function = v1.

Loads the event flags set callback function into the EGCB. Using this service is highly recommended over directly setting internal eg modes, which may result in incorrect settings due to preemption of the current task. Also, direct eg mode setting is not possible in umode under SecureSMX.

Example

```
EGCB  ega;
smx_EventGroupSet(ega, SMX_ST_CBFUN, ega_cbfun);

void ega_cbfun(EGCB_PTR eg)
{
    /* perform callback function */
}
```

This example loads ega_cbfun() into the EGCB. ega_cbfun is called whenever a flags set or pulse operation occurs. This can be used to implement multiple waits.

smx_EventQueue

See the smx User's Guide, Event Queues chapter for usage information and more examples.

smx_EventQueueClear

BOOLEAN smx_EventQueueClear (EQCB_PTR eq)

Type SSR

Summary Clears an event queue.

Compl None

Parameters

eq	Event queue to clear.

Returns

TRUE	Event queue cleared.
FALSE	Error.

Errors SMXE_INV_EQCB Invalid event queue handle.

Descr Resumes all tasks waiting at eq with FALSE return values and deactivates their timeouts. This call would normally be used in a recovery situation, such as starting event processing over.

If the current task is not locked, it may be preempted by a higher priority task that was waiting at eq.

Example

```
EQCB_PTR eq;
smx_EventQueueClear(eq);
```

smx_EventQueueCount

BOOLEAN smx_EventQueueCount (EQCB_PTR eq, u32 count, u32 timeout=0)

Type Limited SSR — tasks only

Summary Suspends the current task on eq for count number of events. Fails if timeout ticks elapse before count events occur.

Compl smx_EventQueueSignal()

Parameters

eq	Event queue to wait at.
count	Number of events to wait for.
timeout	Timeout in ticks or msec if |SMX_FL_MSEC.

Returns

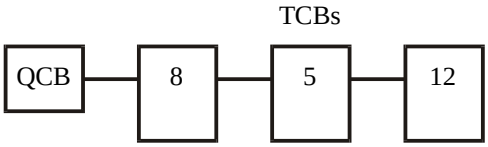
TRUE	Count completed.
FALSE	Error, zero count or timeout, or timed out.

Errors SMXE_OP_NOT_ALLOWED Called from an LSR or timeout == 0.

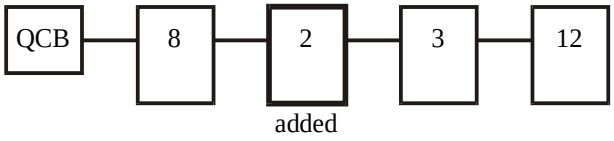
SMXE_INV_EQCB	Invalid event queue handle.
SMXE_BROKEN_Q	Broken queue.

Descr If count is 0 returns TRUE and continues the current task. If it is nonzero, the current task is suspended on eq until it has been signaled count times or for timeout ticks. Then the task is resumed with TRUE or FALSE, respectively.

To enqueue the current task, the differential count of each task already enqueued in eq, is subtracted, in order, from count until the result would be less than 0 or the end of the queue has been reached. The current task is enqueued just ahead of this point or at the end of the queue. The calculated differential count is loaded into the sv field of the current task's TCB and it is subtracted from the differential count of the following task, if there is one. For example, if the event queue looks like this:



and a task with a count of 10 is enqueued, the event queue will then look like this:



smx_EventQueueCount() can be used to delay a task count ticks. However, it is more efficient to suspend it for count ticks:

```
smx_TaskSuspend(smx_ct, count);
```

Event queues are useful to measure events such as revolutions, objects passing by, etc.. The overhead per signal to eq is small because only the first counter need be decremented.

Notes

1. If called from an LSR, operation aborts and returns to LSR.
2. Clears smx_lockctr if called from a task and timeout != SMX_TMO_NOWAIT.
3. The in_evq TCB flag is set to indicate that the task is in an event queue.

Example 1

```
#define SEC   SB_TICKS_PER_SEC

EQCB_PTR  msg_rec;
TCB_PTR  statask;
XCB_PTR  out_port1, in_port1, pool;
```

```
void receiveMain(u32)
{
    MCB_PTR     msg;
    smx_TaskStart(statask);
    while (msg = smx_MsgReceive(in_port1, SMX_TMO_INF))
    {
        /* Process msg */
        smx_EventQueueSignal(msg_rec);
    }
}

void stataskMain(u32)
{
    u8* mbp;
    MCB_PTR status_msg;

    while (1)
    {
        status_msg = smx_MsgGet(pool, &mbp, 0);
        if (smx_EventQueueCount(msg_rec, 8, SEC))
            *mbp = OK;
        else
            *mbp = LOW;
        smx_MsgSend(status_msg, out_port1, 0, NO_REPLY);
    }
}
```

If 8 messages are received in less than a second, OK status is returned. Otherwise LOW status is returned.

smx_EventQueueCountStop

void smx_EventQueueCountStop (EQCB_PTR eq, u32 count, u32 timeout=0)

Type Limited SSR — tasks only

Summary Same as smx_EventQueueCount() except that smx_ct is always stopped, then restarted when it is time for it to run.

Compl smx_EventQueueSignal()

Parameters

eq	Event queue to wait at.
count	Number of events to wait for.
timeout	Timeout in ticks or msec if |SMX_FL_MSEC.

Errors

SMXE_OP_NOT_ALLOWED	Called from an LSR or timeout == 0.
SMXE_INV_EQCB	Invalid event queue handle.
SMXE_BROKEN_Q	Event queue is broken.

Descr See smx_EventQueue() for operational description. smx_ct always stops, then restarts instead of resuming. Pass or fail is returned via the parameter in taskMain(par), when the task restarts.

Notes
1. If called from an LSR, aborts operation and returns to LSR.
2. Clears smx_lockctr if called from a task, since it always stops.

TaskMain void task_main(BOOLEAN par)

par
TRUE Count completed.
FALSE Zero count or timed out.

Example

```
#define SEC   SB_TICKS_PER_SEC

EQCB_PTR  msg_rec;
TCB_PTR     statask;
XCB_PTR     out_port1, in_port1, pool;

void receiveMain(u32)
{
      MCB_PTR  msg;
      smx_TaskStart(statask, 0);
      while (msg = smx_MsgReceive(in_port1, SMX_TMO_INF))
      {
            /* Process msg */
            smx_EventQueueSignal(msg_rec);
      }
}

void stataskMain(BOOLEAN all_msgs_rec)
{
      u8*  mbp;
      MCB_PTR  status_msg;

      if (all_msgs_rec)
      {
            status_msg = smx_MsgGet(pool, &mbp, 0);
            if (all_msgs_rec)
                  *mbp = OK;
            else
                  *mbp = LOW;
            smx_MsgSend(status_msg, out_port1, 0, NO_REPLY);
      }
      smx_EventQueueCountStop(msg_rec, 8, SEC);
}
```

This example is equivalent to that shown for smx_EventQueueCount(). The statask task is started with all_msg_rec == 0 so it will wait at the msg_rec event queue and stop. When 8 messages have been received, the statask task will restart with all_msg_rec == 1.

smx_EventQueueCreate

EQCB_PTR smx_EventQueueCreate (const char* name=NULL , EQCB_PTR* eqhp=NULL)

Type SSR

Summary Creates an event queue.

Compl smx_EventQueueDelete()

Parameters

name	Name to give event queue, NULL for none.
eqhp	Event queue handle pointer (see hp note in Notes and Restrictions) .

Returns

handle	Event queue created.
NULL	Event queue not created due to Error.

Errors

SMXE_OUT_OF_EQCBS	Out of event queue control blocks.

Descr Gets a queue control block from the EQCB pool and initializes it as an EQCB. Loads name into it. Returns the EQCB address as the event queue handle.

Example

```
EQCB_PTR SignalsEQ;

void appl_init(void)
{
    SignalsEQ = smx_EventQueueCreate("SignalsEQ");
}
```

In this example, an event queue is set up to count signals.

smx_EventQueueDelete

BOOLEAN smx_EventQueueDelete (EQCB_PTR* eqhp)

Type SSR

Summary Deletes an event queue.

Compl smx_EventQueueCreate()

Parameters

eqhp	Event queue handle pointer.

Returns

TRUE	Event queue deleted or *eqhp == smx_nullcb
FALSE	Error.

Errors

SMXE_INV_EQCB	Invalid event queue handle.
SMXE_INV_PAR	eqhp == NULL.

Descr Deletes an event queue created by smx_EventQueueCreate(). Resumes all waiting tasks with FALSE return values and clears their timeouts. Then clears the EQCB, releases it to the EQCB pool, and sets *eqhp == smx_nullcb so it cannot be used again. If *eqhp == smx_nullcb, aborts with TRUE.

Example

```
EQCB_PTR RotationsEQ;

smx_EventQueueDelete(&RotationsEQ);
```

smx_EventQueueSignal

BOOLEAN smx_EventQueueSignal (EQCB_PTR eq)

Type SSR

Summary Signals an event queue. The first waiting task waiting may be resumed or restarted.

Compl smx_EventQueueCount(), smx_EventQueueCountStop()

Parameters eq Event queue to signal.

Returns TRUE Signal sent.
FALSE Signal not sent due to error.

Errors SMXE_INV_EQCB Invalid event queue handle.

Descr If eq is an event queue and it is not empty, decrements the first task's count in task->sv. If the resulting count is zero, resumes the first task with TRUE, clears its in_evq flag, and its timeout. Does the same for all other tasks with 0 differential counts. When there are no tasks left in eq, sets eq->fl = NULL and eq->tq = 0.

Example 1

```
LCB_PTR    revLSR;
EQCB_PTR   revs;
TCB_PTR    wheel_task;

void revISR(void)
{
    smx_LSR_INVOKE(revLSR, 0);
}

void revLSRMain(u32 par)
{
    smx_EventQueueSignal(revs);
}
```

```
void wheel_task_main(u32)
{
	while (smx_EventQueueCount(revs, 10, 100)
	{
		/* perform N revs operation */
	}
}
```

Each time a wheel turns, it causes a pulse, which triggers an interrupt causing revISR() to run. revISR() invokes revLSR, which signals the revs event queue. wheel_task runs every 10 revolutions.

smx_Heap

See the smx User's Guide, Heaps chapter for usage information and more examples, and see the eheap User's Guide for more detailed information.

The following heap services are implemented via smx shell porting functions in xheap.c that call corresponding *eheap* services. (eheap is an RTOS-agnostic heap developed for embedded systems.) They meet the ANSI C Standard for malloc(), free(), realloc(), and calloc() and offer many additional services. The smx porting functions provide the following:

1. smx API.
2. Access protection via a mutex per heap.
3. Optional event logging in the smx event buffer.
4. Conversion of eheap errors to smx errors and handling via smx_EM().

smx heap services are not SSRs, hence multiple heaps can be accessed simultaneously by different tasks. For systems without multiple heaps, the heap number, hn, defaults to 0. Access protection can be omitted by directly calling eheap services. Access protection is normally required in a multitasking environment.

eh_hvp[hn] is the heap variable pointer for heap hn. Each heap has its own eheap variable, EHV, structure, which is defined in eheap.h. This structure contains all information needed to control a heap. "eh_hvp[hn]->" has been omitted for heap variables in the following descriptions, but it is required in code that accesses heap variables.

The eheap User's Guide has information concerning how to configure a heap for best size and performance. This is considered advanced information that is needed only when it becomes necessary to optimize a heap and thus is not included in the smx manuals. The following service descriptions are intended to provide sufficient information for use with smx.

smx_HeapBinPeek

u32 smx_HeapBinPeek (u32 binno, EH_PK_PAR par, u32 hn=0)

eheap eh_BinPeek()

Type Mutex-protected function

Summary Returns the current value of the parameter specified for the heap bin specified by binno.

Parameters

binno	Bin number.
par	What to return.
hn	Heap number.

Returns

value	Value of par.
-1	Error.

Errors SMXE_INV_PAR Invalid parameter

Descr Used to obtain information about a heap bin. binno is the bin number. The parameter, par, is of type EH_PK_PAR. Available parameters are:

EH_PK_COUNT	Number of chunks in bin.
EH_PK_FIRST	Address of first chunk in bin, NULL if empty.
EH_PK_LAST	Address of last chunk in bin, NULL if empty.
EH_PK_SIZE	Minimum chunk size for bin.
EH_PK_SPACE	Total free space in bin.

smx_HeapBinPeek() reports SMXE_INV_PAR and returns -1, if par is not one of the above or if binno is not valid. 0 or NULL can be a valid return for some parameters. This service is recommended over directly reading bin parameters, because the latter can result in incorrect readings due to preemption by other tasks. Also, bin parameters cannot be directly read in umode under SecureSMX.

Example

```
CCB_PTR cp;

cp = (CCB_PTR)smx_HeapBinPeek(14, EH_PK_FIRST);
```

This returns a pointer to the first chunk in bin 14.

smx_HeapBinScan

BOOLEAN smx_HeapBinScan (u32 binno, u32 fnum, u32 bnum, u32 hn=0)

eheap eh_BinScan()

Type Mutex-protected function

Summary Scans forward through free bin list of binno for broken links and fixes what it can. Scans backward to fix broken forward links.

Parameters

binno	Bin to scan.
fnum	Number of chunks to scan forward per run.
bnum	Number of chunks to scan backward per run.
hn	Heap number.

Returns

TRUE	Done or unfixable error encountered.
FALSE	Call again to continue scanning.

Errors

SMXE_HEAP_BRKN	The free bin list is broken and cannot be fixed.
SMXE_HEAP_FIXED	A broken link in the free bin list has been fixed.
SMXE_INV_PAR	Invalid parameter.

Descr smx_HeapBinScan() scans the free bin list of bin binno and fixes broken links that it finds or reports SMXE_HEAP_BRKN if a link is unfixable. Normally it is called from a heap manager that runs periodically and scans fnum chunks forward each time. Scans are broken into runs, to permit higher priority tasks to access the heap and not miss their deadlines. If binno is out of range, or if either fnum or bnum is 0, SMXE_INV_PAR is reported and TRUE is returned.

A global pointer, bsp, points at the next chunk to scan, at the start of a run. If it is NULL, a new scan begins from the bin free forward link, ffl. bsp is set to NULL by smx_HeapInit() or when a bin scan completes. Repetitively calling smx_HeapBinScan() each time it returns

FALSE, results in moving through the bin's free chunk list, fnum chunks at a time, until the end of the list is reached and TRUE is returned.

Notes

1. Because it is expected to run frequently, smx_HeapBinScan() makes no entries in the event buffer, other than those due to reported errors or fixes.
2. Whenever a fix is made, EH_HEAP_FIXED is reported, and the scan continues.

Example

```
void smx_HeapManager(void)
{
    static u32  i = 0;
    …
    if (smx_HeapBinScan(i, 2, 10))
        i = i == eh_hvp[hn]->top_bin ? 0 : i+1;
    ...
}
```

This is an example of bin scanning from smx_HeapManager(). smx_HeapBinScan() is called once per pass through smx_HeapManager(), which is called every HEAP_MGR_CNTs from IdleMain(). It scans 2 chunks, at a time. When a bin is finished, smx_HeapBinScan() returns TRUE, and i is incremented to scan the next larger bin. If the top bin has just been scanned, i is cleared and scanning starts over at bin 0.

If the heap has 20,000 free chunks in bins it will take 10,000 passes of idle to scan all bins. If idle runs an average of 100 times per second, it will take 100 seconds to scan all bins. This is probably often enough. If not, fnum can be increased. Note that a backward scan will cover 10 chunks per run. This is because the backward scan is both faster and more urgent since a broken forward link has been found.

If smx_HeapBinScan() fixes a break, it reports SMXE_HEAP_FIXED, which is recorded by smx_EM, and normal operation continues. If it cannot fix a break, it reports SMX_HEAP_BRKN. This is treated as an irrecoverable error by smx_EM(), which calls smx_EMExitHook(), which should initiate recovery. Logging heap fixes into EVB is helpful because it might indicate a bug, a hardware problem, or malware.

smx_HeapBinSeed

BOOLEAN smx_HeapBinSeed (u32 num, u32 bsz, u32 hn=0)

eheap eh_BinSeed()

Type Mutex-protected function

Summary Gets a big enough chunk from heap hn to divide into num chunks for blocks of size bsz and puts them into the correct bin for their size.

Parameters

num	Number of blocks.
bsz	Size of each block, in bytes.
hn	Heap number.

Returns

TRUE	Blocks seeded.
FALSE	Block not seeded due to error.

Errors Same as smx_HeapMalloc() and smx_HeapFree().

Descr This service is used to seed a bin with num chunks having *block size*, bsz. The bin is not specified because it depends upon the *chunk size*. The chunk size = bsz + chunk overhead, EH_CHK_OVH. If debug mode is OFF, EH_CHK_OVH = inuse CCB size = 8 bytes. If debug mode is ON, EH_CHK_OVH = chunk debug control block, CDCB, size + 8*EH_NUM_FENCES - 8.

smx_HeapBinSeed() shares internal subroutines with smx_HeapMalloc() and smx_HeapFree() and thus returns the same errors that they do.

Notes 1. Due to the way smx_HeapMalloc() works, the big chunk may be bigger than necessary. As a consequence the last chunk may be slightly larger than the others and thus might be put into a higher bin.

2. This function performs seeding based upon desired block size. However, bins are based upon chunk size. Hence a correction is necessary to pick the correct bin.

Example

```
u32  trgt_num[sizeof(heap0_binsz)/4];

for (bin = 0; bin <= top_bin; bin++)
{
    if( (n = smx_HeapBinPeek(bin, EH_PK_COUNT)) <= trgt_num[bin])
    {
        bsz = smx_HeapBinPeek(bin, EH_PK_SZ) - EH_CHK_OVH;
        num = trgt_num[n] – n;
        smx_HeapBinSeed(num, bsz);
    }
}
```

This function compares bin contents to a target size for each bin from 0 to the top_bin and seeds the bin if necessary to bring it up to the target number. Bin seeding might be used to get the heap off to a good start during initialization.

smx_HeapBinSort

BOOLEAN smx_HeapBinSort (u32 binno, u32 fnum, u32 hn=0)

eheap eh_BinSort()

Type Mutex-protected function

Summary Sorts a large bin’s free chunk list by increasing chunk size.

Parameters

	binno	Bin number.
	fnum	Number of chunks to sort per run.
	hn	Heap number.

Returns

TRUE	Bin sort has been completed, was not needed, or was aborted due to an error.
FALSE	Call again to continue sorting this bin.

Errors SMXE_INV_PAR fnum is 0.

Descr smx_HeapMalloc() and the other heap allocation services take the first large-enough chunk from a large bin. If the bin's free chunk list is sorted by increasing size, this will be the best-fit chunk in the bin. Thus large-bin sorting helps improve allocation times from large bins. Also, since the allocated chunk is best fit, splitting and thus fragmentation is reduced.

This service is used to put chunks in order, by increasing size in large-bin free lists. A run consists of fnum sorting loops. fnum is chosen to be small enough so that higher priority tasks needing access to this heap do not miss their deadlines, yet large enough so that bins will usually be sorted when needed. Bin sorting is normally done during idle time.

The bin sort map, bsmap, has a bit per bin. The bit for a bin is set if smx_HeapFree() puts a chunk into the bin's free list. When the chunk is larger than the first chunk in the bin, it is put at the end of the list and the bsmap bit is set for the bin. Otherwise the chunk is put at the start of the bin's free list and the bsmap bit is not set. Small bins never need sorting, hence their bsmap bits are never set. Therefore, bsmap shows only large bins that need sorting.

If binno is less than or equal to the top bin number and its bsmap bit is ON, that bin is selected. Else if binno is greater than the top bin number, the smallest bin having its bsmap bit ON is selected. Calling smx_HeapBinSort() repetitively until it returns TRUE results in sorting the selected bin. The bsmap bit is cleared on the first pass of sorting a bin.

Each time fnum chunks have been sorted, smx_HeapBinSort() gives up the heap mutex so a higher-priority heap operation can run.

A bin's bsmap bit is reset when a sort begins, and csbin stores the bin number being sorted, between runs. If a preempting free sets the bit, due to putting a chunk at the end of the bin, the sort is aborted and restarted on the next run. If a preempting malloc takes a chunk from the bin, it also sets the bin's bsmap bit, causing the sort to start over. Starting over is not detrimental to a sort, because any sorting already done is preserved. Otherwise each run starts from where the last run left off.

Notes 1. Heap sorting need not be perfect. Taking a somewhat larger chunk than necessary due to imperfect sorting is not likely to be significant.

2. Because it is expected to run frequently, smx_HeapBinSort() makes no entries in the event buffer, other than those due to reported errors or fixes.

Example 1

```
void smx_HeapManager(void)
{
    for(i = first_large_bin, i <= top_bin, i++)
    {
        while (!smx_HeapBinSort(i, 4) {}
    }
    delay(n);
}
```

Example 2

```
void smx_HeapManager(void)
{
    while (!smx_HeapBinSort(top_bin + 1, 4)
    {
        while (!smx_HeapBinSort(top_bin + 1, 4) {}
    }
    delay(n);
}
```

Example 1 one shows going methodically through all large bins every n time units. Example 2 shows where the first smx_HeapBinSort() finds the smallest unsorted bin, if any, and calls smx_HeapBinSort() repetitively to sort that bin. It then finds the next smallest unsorted bin and continues until all bins have been sorted, then waits n time units to start over. In both cases, smx_HeapBinSort() is called repetitively until it returns TRUE, meaning that the bin is fully sorted. Example 2 is clearly more efficient than example 1.

smx_HeapCalloc

void* smx_HeapCalloc (u32 num, u32 sz, u32 an=0, u32 hn=0)

eheap eh_Calloc()

Type Mutex-protected function

Summary Allocates space for an array of num elements of sz bytes from the heap and clears all elements. See smx_HeapMalloc() for details concerning allocations.

Compl smx_HeapFree()

Parameters

num	Number of elements.
sz	Size of each element in bytes.
an	Alignment number (block alignment = 2^an bytes).
hn	Heap number.

Returns

pointer	to allocated array.
NULL	Array not allocated due to error.

Errors Same as smx_HeapMalloc()

Descr Allocates a single block of memory from the heap of (num * sz) bytes with fill mode OFF. The contents of the block are cleared, fill mode is restored, and a pointer to the block is returned. It is important to note that only one heap block is allocated, and therefore, the blocks in the array cannot be individually freed. This service shares internal subroutines with smx_HeapMalloc()and thus returns the same errors that it does.

Example

```
#define NUM_RECS 10

typedef struct  {
      u32 field1;
      u32 field2;
} REC;

REC *rp;
u32   i, error;

void array_op(void)
{
      if (rp = (REC*)smx_HeapCalloc(NUM_RECS, sizeof(REC)))
            for (i = 0; i < NUM_RECS; i++, rp++)
            {
                  rp->field1 = i;
                  rp->field1 = 2*i;
            }
      else
            /* report error */
}
```

smx_HeapChunkPeek

u32 smx_HeapChunkPeek (void* vp, EH_PK_PAR par, u32 hn=0)

eheap eh_ChunkPeek()

Type Mutex-protected function

Summary Returns the current value of the parameter specified for a chunk in the heap, given a pointer to either the chunk or to the block in it.

Parameters

	vp	Chunk pointer (cp) or block pointer (bp).
	par	What to return.
	hn	Heap number.

Returns

value	Value of par.
-1	Error.

Errors

SMXE_INV_PAR	Invalid parameter.
EH_WRONG_HEAP	vp is not in heap n range or is not 4-byte aligned.

Descr Used to return information about heap chunks. The parameter, par, is of type EH_PK_PAR. Permitted values are:

EH_PK_BINNO	Chunk bin number (0 if not free, or not in a bin).
EH_PK_BP	Data block pointer determined from cp (0 if free).
EH_PK_CP	Chunk pointer determined from bp (0 if free).
EH_PK_NEXT	Address of next chunk in the heap.
EH_PK_NEXT_FREE	Address of next chunk in this bin (0 if last chunk, not in bin, or not free).

EH_PK_ONR	Chunk owner (0 if not a debug chunk).
EH_PK_PREV	Address of previous chunk in heap.
EH_PK_PREV_FREE	Address of previous chunk in bin (0 if first chunk, not in bin, or not free).
EH_PK_SIZE	Chunk size.
EH_PK_TIME	Time chunk allocated (0 if not debug chunk).
EH_PK_TYPE	Chunk type (free == 0, inuse == 1, debug == 3).

smx_HeapChunkPeek() returns -1 and reports SMXE_INV_PAR, if par is not one of the above values. If a chunk is inuse, it cannot be in a bin, thus 0 is returned. Since 0 is a valid bin number, the chunk should be tested for free. Care must be taken that vp is a valid chunk pointer in all cases, unless par == EH_PK_CP, in which case it must be a valid data block pointer.

Using this service is recommended over directly reading chunk parameters. The latter may result in incorrect readings, due to preemption by another task or due to attempting to read an invalid field for the chunk type. Also, chunk parameters cannot be directly read in umode under SecureSMX. It usually is best to read the chunk type first to make sure that the expected chunk information is actually available. It is also advisable to check that the return value is not -1 before using it.

Example

```
#define DEBUG 3

u8* bp;
CCB_PTR cp;
int time = 0;

if (cp = (CCB_PTR)smx_HeapChunkPeek(bp, EH_PK_CP))
    if (smx_HeapChunkPeek(cp, EH_PK_TYPE) == DEBUG)
        time = smx_HeapChunkPeek(cp, EH_PK_TIME);
```

Starting with a block pointer, this example show how to get the chunk pointer, cp, then determine when the block was allocated, if it is in a debug chunk.

smx_HeapExtend

BOOLEAN smx_HeapExtend (u32 xsz, u8* xp, u32 hn=0)

eheap eh_Extend()

Type Mutex-protected function

Summary Adds a memory extension to the heap.

Parameters

xsz	Extension size, in bytes.
xp	Extension pointer.
hn	Heap number.

Returns

TRUE	Heap extended.
FALSE	Heap not extended due to error.

Errors SMXE_INV_PAR xsz is zero or xp is not above current heap.

Descr smx_HeapExtend() is used to extend the heap to additional memory space. xsz is the size of the additional space and xp is a pointer to the start of it. This space can come from any RAM that is above the current heap. If not, smx_HeapExtend() reports SMXE_INV_PAR, and returns FALSE. This is also the case if xsz == 0. Otherwise, xsz is increased to 16 or set to the next 8-byte boundary and xp is moved up to the next 8-byte boundary, if necessary.

smx_HeapExtend() handles both the case where the extension is adjacent to the top of the current heap and the case where there is a gap in between. In both cases, ec (end chunk) is moved to the top of the extension. In the adjacent case, the extension is merged with the top chunk, tc, and the merged chunk becomes the new tc. In the gap case, an artificial inuse chunk is created from the old ec to cover the gap and the extension becomes the new tc. The old tc is freed to a bin. tc and the freed chunk are filled if fill mode is ON. Then tcp and hsz are updated and TRUE is returned.

Example

```
#define HEAP_EXT  4096

u8*  xp = 0xC0200000;
BOOLEAN  ok;

if (smx_errno = SMXE_INSUFF_HEAP)
{
      ok = smx_HeapExtend(HEAP_EXT, xp);
}

if (ok)
      /* retry allocation */
```

This example shows extending the heap by 4096 bytes in order to recover from an SMXE_INSUFF_HEAP error. In this case, 0xC0200000 is the start of free DRAM.

smx_HeapFree

BOOLEAN smx_HeapFree (void* bp, u32 hn=0)

eheap eh_Free()

Type Mutex-protected function

Summary Frees a block to the heap or heap block pool that was previously allocated from the heap or heap block pool.

Compl smx_HeapMalloc(), smx_HeapCalloc(), and smx_HeapRealloc()

Parameters

bp	Pointer to block to free.
hn	Heap number.

Returns

TRUE	Block freed or already free.
FALSE	Block not freed due to an error.

Errors

SMXE_HEAP_ERROR	Block is already free.
SMXE_INV_CCB	Forward or backward link is out of range.
SMXE_INV_PAR	Derived cp is out of range or not 8-byte aligned

Descr

Frees the block pointed to by bp back to the heap. If bp is NULL, no operation is performed and TRUE is returned, per the ANSI C standard. If bp is not in heap hn range, SMXE_INV_PAR is reported and FALSE is returned.

If EH_BP (Block Pool enable) and bp is less than first heap chunk pointer, fhcp, then the block pointed to by bp is freed to either the 8-byte pool or to the 12-byte pool. The *block pointer control block pointer, bpcbp,* for hn, points to an array of two *block pool control blocks, BPCBs* (see eheap.h). The BPCB is selected depending upon bp. The pn field in a BPCB points to the first block in the free block linked list. The freed block is put at the start of this list, pn is updated to point to it, and TRUE is returned. This operation is very fast compared to a heap free. If bpcbp == NULL, the operation is aborted and FALSE is returned.

A double free is detected by testing the inuse bit of the word before bp, which is the last word of the chunk control block. If 0, EH_HEAP_ERROR is reported and FALSE is returned. This test is not 100% effective because the chunk may have already been reallocated and thus pass the test. In this case, the reallocated block will be freed, in error.

bp is converted to the *chunk pointer, cp,* for the chunk it is in, and a pointer to the previous chunk, the *prechunk pointer, pcp,* is derived from cp. If either pointer is out of heap range for heap hn, FALSE is returned and EH_INV_CCB is reported. If EH_SS_MERGE (Spare Space Merge), the prechunk is inuse, and it has spare space at the end, the spare space is merged with the freed chunk.

If merging is enabled by mode.fl.cmerge == ON, smx_HeapFree() merges the prechunk if it is free and merges the postchunk if it is free. Chunks in bins are removed from their bins before merging them. If the postchunk is dc the merged free chunk is put into dc and dcp is updated; if the postchunk is tc the chunk is put into tc and tcp is updated. Otherwise, the correct bin for the chunk size is determined and the chunk is put into that bin. heap_used is reduced by the size of the freed chunk.

Notes

1. If chunk filling is enabled (mode.fl.fill == ON) a free block is loaded with the EH_FREE_FILL pattern. However, dc or tc is loaded with the EH_DTC_FILL pattern. This greatly increases the time required to free a block and should be used only to assist debugging.

2. If either of the scan pointers, hsp or hfp, was pointing at the freed chunk and it was merged with a prechunk or spare space, the pointer is backed up to the start of the new chunk. If a chunk is put at the end of a large bin, the bsmap bit for that bin is set, indicating that the bin needs to be sorted.

Example

```
void function(void)
{
    void *dp;

    dp = smx_HeapMalloc(100);
    /* use temporary block of memory via dp */
    smx_HeapFree(dp);
}
```

This example gets a block of 100 bytes from the heap 0, uses it, then frees it back to the heap.

smx_HeapInit

u32 smx_HeapInit (u32 sz, u32 dcsz, u8* hp, EHV_PTR vp, u32 mode)

eheap eh_Init()

Summary Initializes heap starting at hp with heap variables structure selected by vp.

Parameters

sz	Size of the heap, in bytes.
dcsz	Donor chunk size. 0 means no donor chunk.
hp	Start of heap pointer.
vp	Pointer to heap variables structure.
mode	Heap mode flags.

Mode Flags

EH_CM	chunk merge*	
EH_DBM	debug mode*	
EH_FILL	fill*	
EH_AM	automerge*	
EH_HFR	heap failure report	
EH_AR	auto recover	
EH_ED	error detection excluding allocations and frees	
EH_EDA	error detection all	
EH_EM	error manager	
EH_PRE	preemption protection	
EH_NORM	(EH_AM | EH_EDA | EH_EM | EH_PRE)	normal operation
EH_DBOP	(EH_NORM | EH_FILL | EH_HFR)	debug operation

Returns

hn	Heap number (means heap has been initialized).
-1	Heap not initialized due to an error or it was already initialized.

Errors

SMXE_ALREADY_INIT	Heap has already been initialized.
SMXE_INV_PAR	sz or hp is invalid.
SMXE_TOO_MANY_HEAPS	SMX_NUM_HEAPS have already been initialized.

Descr A heap must be initialized before it can be used. If C++ is not being used in an application or in a partition of an application using a dedicated heap, smx_HeapInit() can be called from the initialization code for the application or the partition before any other heap calls are made. smx_HeapInit() returns the heap number, hn, which must be used in all subsequent heap calls for that heap, unless hn = 0, which is the default value.

If C++ is being used in an application or in a partition of an application that uses a dedicated heap, smx_HeapInit() must be called after the compiler startup does data init, but before C++ initializers run. This is because data init clears RAM and C++ initializers normally make heap allocation calls to create C++ objects. Some compilers (e.g. IAR EWARM) provide a window to do this, but other compilers do not.

To deal with the latter case, code is put into smx_HeapMalloc() to call a user-written function, smx_HeapsInit() if eh_hvpn == 0 – i.e. no heaps have been initialized. This function

must initialize all heaps associated with C++ code. This is unfortunate because the code to test eh_hvpn becomes overhead for every allocation thereafter.

hp can point anywhere in RAM and sz can be any desired size >= 32 bytes. Typically a main heap is allocated from SRAM or DRAM, and small dedicated heaps may be allocated from it. In some systems the main heap may be allocated from DRAM, and small fast heaps may be allocated from SRAM. Allocation of space for a heap is done in the linker command file.

eheap maintains an array of pointers, eh_hvp[EH_NUM_HEAPS]. Each pointer points to the heap variable structure (EHV) for heap hn, where hn is the index into eh_hvp. Hence, eh_hvp[hn]-> is used to access heap hn variables. "eh_hvp[h]->" is omitted when referring to heap variables in this manual, but it must be included in code. Space for each heap variable structure is allocated by application code and certain fields must be set before calling smx_HeapInit() — see the example below. When a heap has been successfully initialized, its heap number, hn, is returned and mode.fl.init is turned ON. Thereafter, hn must be used for all accesses to that heap, except h0, which is the default or *main heap*.

If vp is NULL, -1 is returned and nothing is done. If sz < 32 or hp is NULL, SMXE_INV_PAR is reported; if eh_hvp[] is full, SMXE_TOO_MANY_HEAPS is reported; if mode.fl.init is ON, SMXE_ALREADY_INIT is reported. In all cases -1 is returned and nothing is done.

If they are not multiples of 8, sz is adjusted to the next lower multiple of 8 and hp is adjusted to the next higher multiple of 8. This is done so that the heap and all chunks in it will be 8-byte aligned.

Following initialization, the heap consists of four chunks: start chunk (sc), donor chunk (dc), top chunk (tc), and end chunk (ec). sc and ec are inuse chunks with no data. They are each 8 bytes in size and linked together. dc is a free chunk, which initially contains dcsz bytes. tc is a free chunk, which initially contains the remaining free space of the heap = sz – dcsz - 16. dc normally is much smaller than tc; it is the source for small chunks. If dcsz < 24, dc becomes a free chunk with no space for data and mode.fl.use_dc is turned OFF. tc is the source for large chunks.

If EH_BP and bpcbp is not NULL, space is allocated from the heap for 8-byte and 12-byte block pools at the bottom of the heap between sc and dc and the pools are initialized. bpcbp points at an array of block pool control blocks, BPCBs (see eheap.c). Each BPCB has a num_blks field, which is loaded by the application. If num_blks is 0, no pool is created. Otherwise, a num_blks pool is created and its pool control block is initialized.

smx_HeapInit() loads pi = sc and px = ec. It initializes the mode field so that cmerge, debug and fill flags are OFF and other flags are ON. If EH_PRE, preemption protection is enabled and eh_hvp[n]->pre is set, a heap mutex is created for the heap, and the mutex handle is loaded into the mtx field. This mutex is used to control access to all heap functions for this heap.

If mode == 0, hmtx is not created, and heap functions are not protected by a mutex for this heap. This mode of operation is intended for partition heaps in which preemption of heap operations is prevented by other means. This reduces the overhead for the heap operations.

smx_HeapInit() also initializes the bins and other heap variables. If hmode.fl.fill is ON, dc and tc are filled with the EH_DTC_FILL pattern.

The mode flags enable corresponding heap operations, if set. If a flag is present in the mode argument, it is set in eh_hvp[n]->mode; otherwise, it is reset in eh_hvp[n]->mode. The flags marked with * can also be set and reset with smx_HeapSet().

See the Setup chapter of the eheap User's Guide for detailed information on setting up and initializing heaps.

Example

```
u32 const hm_binsz[] =

/*bin  0    1      2      3      4      5      6      7      8      9     10     11 */
     {24,  32,   40,   48,   56,   64,   72,   80,   88,   96,   104,  112, \
/*bin 12    13    14     15     16      17     18     19      20      21      22      23 */
     120,  128,  256,  384,  512,  640,  768,  896,  1024, 1152, 1280, 1408, \
/*bin 24      25      26      27      28   end */
     1536, 1664, 1792, 1920, 2048, -1};

/* get heap space for hm allocated in linker command file */
u8*  hmsa = (u8*)__section_begin("hm"); /* heap main starting address */
u32 hmsz = (u32)__section_size("hm");   /* heap main size */
u32 hmdcsz = hmsz/8;  /* heap main donor chunk size */
EHV hmv;  /* heap main variables structure */
HBCB hm_bins[(sizeof(hm_binsz)/4)-1];   /* heap main bins */
u32 hmn;  /* heap main number */

memset((void*)&hmv, 0, sizeof(EHV));  /* clear hmv */
hmv.bszap = (u32*)hm_binsz;
hmv.binp  = (HBCB*)hm_bins;
hmn = smx_HeapInit(hmsz, hmdcsz, hmsa, &hmv, (EH_EDA | EH_EM | EH_PRE));
hmv.name  = "hm";
```

The above example shows creating the main heap for a system. Hence, there are a large number of bins. hm_binsz is an array of bin sizes, each being the minimum size for that bin. Note that bins 0 to 12 are the *small bin array, SBA*, for this heap. Each of these bins holds only one size. (hm_binsz/8 – 3) is used as an index and the first free chunk is taken, so access is very fast. Bins 13 to 27 are the *large bin array, LBA*. Each of these bins is spaced 128 bytes apart. Hence, each holds 128/8 = 16 chunk sizes. To get a chunk from one of these bins requires searching for the first large enough free chunk. The bins are normally sorted to make this faster and to allocate a *best-fit chunk*. Bin 28 is the top bin. It holds chunk sizes from 2048 bytes and up.

Space for heap main, hm, is allocated by the linker command file. hmsa and hmsz are obtained, as shown. hmv is statically allocated, as is the hm_bins array of bin pointers. hmv is cleared, two fields in it are preset, then smx_HeapInit() is called to initialize the heap and finally the heap is given a name, "hm". smx_HeapInit() returns the heap number, hmn. This can be used to access the EHV for the heap main via eh_hvp[hmn]->field. Heap numbers are assigned in the order that heaps are created, up to a maximum, EH_NUM_HEAPS, defined in eheap.h. hmv.field can also be used to access the EHV for hm.

Heaps can have any number of bins from 1 to 28. For example:

```
u32 const binsz1[] =
/*bin  0  end */
    {24,  -1};
```

Defines a one bin heap. Being able to adjust the number and the sizes of bins is useful for partition heaps under SecureSMX.

smx_HeapMalloc

void* smx_HeapMalloc (u32 sz, u32 an=0, u32 hn=0)

eheap eh_Malloc()

Type Mutex-protected function

Summary Allocates a block of at least sz bytes from heap hn, aligned on at least a 2^an-byte boundary. Also can perform MPU region block allocations under SecureSMX.

Compl smx_HeapFree()

Parameters

	sz	Minimum size of block to allocate, in bytes.
	an	Alignment number (block alignment = 2^an bytes).
	hn	Heap number.

Returns

bp	Block pointer.
NULL	Insufficient space or error.

Errors

SMXE_INV_PAR	Invalid parameter if sz = 0 or > heap size
SMXE_INSUFF_HEAP	Insufficient space in heap.

Descr Allocates a block of at least sz bytes and aligned on a 2^an-byte boundary from heap hn. The block is contained within a *chunk*. If debug mode is OFF, an *inuse chunk* is allocated; if debug mode is ON, a *debug chunk* is allocated. Chunks are normally hidden from the user. The minimum block size that can be allocated from the heap is 16-bytes. The block size may be larger than sz, if an exact-fit chunk has not been found.

Prior to searching the heap if block pools are present, a block of sz <= 12 and an <= 3 is taken from the 8-byte or 12-byte pool, if the pool is not empty and there is not an alignment mismatch (12-byte blocks can be 4-byte aligned.) Otherwise a heap search starts.

If sz is 0 or > heap size, SMXE_INV_PAR is reported and NULL is returned. If sz is less than 16, it is rounded up to 16. If sz is not a multiple of 8, it is adjusted to the next higher multiple of 8. For example, if sz = 27, it is adjusted to 32.

The search for the needed chunk progresses as follows until it is found: a small chunk is taken from the right-size bin in the small bin array (SBA), the donor chunk (dc), the next occupied bin, or the top chunk (tc). A large chunk is taken from the upper bin for its size, the next occupied bin, or tc. If allocation fails automatic recovery will occur, if enabled (see smx_HeapRecover() discussion). Otherwise EH_INSUFF_HEAP is reported and NULL is returned..

All blocks from the heap are automatically 8-byte (an = 3) aligned. If enabled, chunks with larger alignments can be allocated. The search order is the same as above. In this case, a chunk is chosen if the data block will fit at the next 2^an boundary and still be within the chunk. The CCB is moved up to this boundary and spare space is combined with the prechunk (see smx_HeapFree() discussion). If EH_R is added to the alignment number to form the an parameter (e.g. EH_R + 3), an *MPU region block allocation* will be performed. See the eheap User's Guide, Chapter 4 Operation for more information on both of these types of allocations.

The found chunk is marked inuse and split if its spare space is greater than or equal to EH_MIN_FRAG, defined in eheap.h. The upper part becomes a free chunk. It will be merged into a *free postchunk* (i.e. the next chunk) if mode.fl.cmerge is ON. If the found chunk's spare space is less than EH_MIN_FRAG, its EH_SSP flag is set (bit 2 in blf) and its *spare space pointer, ssp*, is loaded into the last word of the chunk. ssp points to the start of the spare space. See the discussion in smx_HeapFree() for how spare space is used.

If fill mode is enabled, unique fill values are put into the data and spare space areas. These help when viewing the heap via the debugger memory window.

If mode.fl.debug is ON, a debug chunk is allocated instead of an inuse chunk. The CCB is replaced with a CDCB (Chunk Debug Control Block) followed by EH_NUM_FENCES ahead of and after the data block. See the smx User's Guide, Heaps Chapter, Heap Debugging section for more information on debug chunks.

The final chunk size is added to hused, which is used to determine the *high-water mark, hhwm* of heap usage in order to determine if the heap needs more memory. If allocation fails, NULL is returned and EH_INSUFF_HEAP is reported.

Example

```
void* bp;

if (bp = smx_HeapMalloc(204, 5, hm))
{
    /* access block using bp */
    smx_HeapFree(bp);
}
```

Since 204 is not a multiple of 8, the size is increased to 208. A block of 208 bytes, aligned on a 2^5 = 32-byte boundary, is allocated from the main heap. If the main heap is in DRAM and the cache line size is 32 bytes, this alignment will improve access times to the block. When no longer needed, the block is released back to the heap by smx_HeapFree().

smx_HeapPeek

u32 smx_HeapPeek (EH_PK_PAR par, u32 hn=0)

eheap eh_Peek()

Type Mutex-protected function

Summary Returns the current value of the parameter specified for heap hn.

Compl smx_HeapSet()

Parameters

par	What to return.
hn	Heap number.

Returns

value	Value of par.
-1	Error.

Errors SMXE_INV_PAR Invalid parameter.

Descr Used to obtain information about heap, hn. The parameter, par, is of type EH_PK_PAR. Permitted values are:

EH_PK_AUTO	Automatic chunk merge control is enabled.
EH_PK_BS_FWD	Bin scan forward.
EH_PK_DEBUG	Allocate debug chunks.
EH_PK_FILL	Fill blocks, spare space, dc, and tc with unique fill patterns.
EH_PK_HS_FWD	Heap scan forward.
EH_PK_INIT	Heap has been initialized.
EH_PK_MERGE	Merge chunks when freed.
EH_PK_USE_DC	Enable allocation from donor chunk.

smx_HeapChunkPeek() returns -1, and reports SMXE_INV_PAR, if par is not one of the above. Otherwise, it returns the value of the mode (ON or OFF). This service is recommended over directly reading heap modes, because the latter can result in incorrect readings due to preemption by other tasks. Also, heap modes cannot be directly read in umode under SecureSMX.

Example

```
if (smx_HeapPeek(EH_PK_MERGE) )
    /* chunks are being merged, when freed */
else
    /* chunks are not being merged, when freed */
```

This might be used to monitor how automatic merge control is doing or to decide what action to take if a heap failure has occurred.

smx_HeapRealloc

void* smx_HeapRealloc (void* cbp, u32 sz, u32 an=0, u32 hn=0)

eheap eh_Realloc

Type Mutex-protected function

Summary Allocates a new size block from an existing heap block. Preserves existing contents and conforms to the ANSI C Standard. See smx_HeapMalloc() for details concerning allocations.

Compl smx_HeapFree()

Parameters

	cbp	Pointer to block to reallocate.
	sz	New block size.
	an	Alignment number (block alignment = 2^an bytes).
	hn	Heap number.

Returns

	nbp	New block pointer.
	NULL	Insufficient space or error.

Errors Same as smx_HeapMalloc() and smx_HeapFree().

Descr This service is generally used to allow a task to release memory that it no longer needs, without having to get another block and copy the data from the old block to the new block. In this case, time saved by using smx_HeapRealloc() can be substantial.

Alternatively, smx_HeapRealloc() allows getting a larger block, and data in the old block will be automatically copied over to the new block. Since it cannot be preempted, a higher-priority task needing the heap will not be able to run until it finishes. Hence, if working with large blocks, it may be preferable to malloc a larger block, copy the data, then free the smaller block, instead of using smx_HeapRealloc().

Reallocates an existing block pointed to by cbp to a new block of size, sz, and returns a new block pointer, nbp. Can be used to either downsize or upsize the current block at cbp. smx_HeapRealloc() is considerably more complex than the other two heap allocation services. However, it uses smx_HeapMalloc() and smx_HeapFree(), so the same discussion for them concerning size, errors, etc. applies to it.

Per the ANSI C Standard: if cbp == NULL, a block of sz bytes is allocated from the heap; if sz == 0, cbp is freed to the heap. Otherwise, if cbp is not within heap hn range or not 8-byte aligned, SMXE_WRONG_HEAP is reported and NULL is returned. If sz is greater than 0, but less than 16, it is automatically rounded up to 16; if sz is not a multiple of 8, it is rounded up to the next multiple of 8.

The current chunk size is determined and the necessary new chunk size is determined. If mode.fl.debug is OFF the latter will be for an inuse chunk, else it will be for a debug chunk. This is true, regardless of the type of the current chunk, which is being reallocated. Hence, smx_HeapRealloc() can be used to convert an inuse chunk to a debug chunk or vice versa, without losing data in the data block. smx_HeapRealloc() can also be used to increase the alignment of the block. Either of these is likely to require a new chunk.

There are two possibilities for reallocation, due to relative chunk sizes:

current chunk is big enough, then it is split into a new, exact-fit chunk and a new free chunk[1]. The new free chunk is merged with the chunk after[2], if it is free and cmerge is ON. The block pointer returned, nbp, is the same as cbp and the block size is equal to or slightly larger than sz[3]. Note that data up to the new size is preserved and that data above that size is lost.

current chunk is not big enough, then the current chunk is freed. This may result in its being merged with a lower free chunk or a upper free chunk, or both, which could result in a chunk that is now big enough for the new block. However, the odds of that occurring are small, so the new free chunk is put into a bin, and the smx_HeapMalloc() is called to get the best-fit chunk that can be found. Then data is copied from the current block to the new block, if necessary[4], and the new block pointer, nbp, is returned. Also, the unused upper portion of the chunk is split off into a new free chunk, if it is big enough[2].

If a big-enough chunk cannot be found, the preceding free, merge, and bin load operations are reversed, SMXE_INSUFF_HEAP is reported, and NULL is returned. In this case, the initial block is undisturbed and can continue being used via the cbp pointer. Means to recover from this failure are the same as described for smx_HeapMalloc().

In all cases, data is preserved up to the end of the current block or to the end of the new block, whichever is smaller. To ensure this, fill mode is turned OFF, then restored at the end of this service. Thus heap fill is suspended for all smx_HeapRealloc() operations.

Example

```
void *bp, *nbp;

bp = smx_HeapMalloc(200);
/* use 200-byte block via bp */

/* get another 200 bytes */
nbp = smx_HeapRealloc(bp, 400);
/* use 400-byte block via nbp *
```

This example allocates 200 bytes from the heap, uses it for a while, then increases the block size to 400 bytes. When a block is being increased in size, the most likely scenario is that a larger chunk will be allocated elsewhere in the heap, the data from the old block will be copied to the new block, then the old chunk will be freed. In the above example, nbp is unlikely to be the same as bp. Hence, care must be exercised to update any secondary pointers (e.g. read pointer, write pointer, etc.). The contents from byte 0 to byte 199 of the original block are guaranteed to be unchanged, even though the block may have been moved.

[1] There is a limitation on chunk splitting. See discussion in the eheap User's Guide, Operation Chapter, chunk splitting section.
[2] When discussing chunks, "before" and "after" or "lower" and "upper" refer to physical chunk positions.
[3] See discussion in smx_HeapMalloc().
[4] It is possible that the chunk and data block do not move, even though they are larger, in which case block contents are not copied.

smx_HeapRecover

BOOLEAN smx_HeapRecover (u32 sz, u32 num, u32 an=0, u32 hn=0)

eheap eh_Recover()

Type Mutex-protected function

Summary Tries to find enough adjacent free chunks that can be merged to create a chunk large enough for a block of sz bytes with alignment an. See smx_HeapMalloc() for details concerning allocations.

Parameters

sz	Block size needed.
num	Maximum number of chunks to scan.
an	Alignment number (block alignment = 2^an bytes).
hn	Heap number.

Returns

TRUE	Chunk is now available to allocate.
FALSE	Chunk not found.

Errors SMXE_INV_PAR Invalid parameter: sz or num = 0.

Descr This service is intended to recover from a situation where a large chunk cannot be allocated because the heap has been fragmented into too many smaller free chunks. Recovery is possible only if enough free space is found in adjacent free chunks. Otherwise, this service fails and some other means must be used to allocate the needed chunk.

smx_HeapRecover() starts the scan from sc for small chunks or from the top of dc for large chunks. The scan continues until num chunks have been scanned or ec has been reached. If a big-enough chunk can be formed by merging adjacent free chunks, it removes the free chunks (except dc and tc) from their bins and merges them. mode.fl.cmerge, if set, is ignored. If the merged chunk is not dc nor tc, it puts the merged chunk into its proper bin. If the merged chunk is dc, it updates smx_dcp; if the merged chunk is tc, it updates smx_tcp, then returns TRUE.

smx_HeapRecover() does not merge chunks that it cannot use. If successful, smx_HeapRecover() should be followed by retrying the allocation that failed. If mode.fl.auto_rec is ON, this is done automatically and the allocation returns the block in the merged chunk. In this case, recovery is transparent to the application, except that the allocation will take longer than normal and SMXE_HEAP_RECOVER will be reported. In this case, the entire heap is searched. If a big-enough chunk is not found, returns FALSE and the allocation returns NULL and reports SMX_INSUFF_HEAP.

If smx_HeapRecover() is called directly, it will search for num chunks and return FALSE if nothing is found. This is intended to put a limit on search times for very large heaps; it allows application recovery code to try another approach or to simply move on. In this case mode.fl.auto_rec must be OFF. Allocation failure is most likely to occur for large blocks while the heap is still usable for smaller blocks. In time, the large block allocation might be tried again, and it might succeed.

If num expires on a free chunk, the scan continues until a big-enough free space is found, an inuse chunk is found, or the end of the heap is reached. If a big-enough free space is found, the chunks are merged and TRUE is returned.

Example

```
void* bp;
TCP_PTR StoppedTask;

void ProcessTaskMain()   /* for mode.fl.auto_rec = OFF */
{
    while (1)
    {
        if (bp = smx_HeapMalloc(1000, 0, fheap))
        {
            /* process data using bp */
            smx_HeapFree(bp);
        }
        else
            break;
    }
    smx_TaskStart(RecoveryTask, 1000);
    StoppedTask = smx_ct;
}

void RecoveryTaskMain(u32 size)
{
    if (smx_HeapRecover(size, 10000, 0, fheap)
        smx_TaskStart(StoppedTask);
    else
        /* use alternate recovery method */
}
```

In the above example, if smx_HeapMalloc() fails in ProcessTask, RecoveryTask is started with the needed size as a parameter, ProcessTask's handle is saved in StoppedTask, and ProcessTask autostops. When RecoveryTask runs, it calls smx_HeapRecover(), which tests up to 10,000 chunks. If it finds a big-enough chunk it returns TRUE, which restarts ProcessTask. If not, ProcessTask remains stopped while alternate recovery techniques are tried, such as extending fheap, using a different heap, releasing unneeded blocks, or reallocating blocks smaller.

smx_HeapScan

BOOLEAN smx_HeapScan (CCB_PTR cp, u32 fnum, u32 bnum, u32 hn=0)

eheap eh_Scan

Type Mutex-protected function

Summary Scans forward through the heap for errors and makes fixes when it can. Scans backward through the heap to fix broken forward links.

Parameters

cp	Chunk pointer to start scan. Start at smx_hsp, if NULL.
fnum	Number of chunks to scan forward per run.
bnum	Number of chunks to scan backward per run.

	hn	Heap number.

Returns

TRUE	Stop scanning – done or unfixable error encountered.
FALSE	Continue scanning.

Errors

SMXE_HEAP_BRKN	Heap cannot be fixed.
SMXE_HEAP_FENCE_BRKN	Broken fence found (fixed in release version).
SMXE_HEAP_FIXED	A heap fix was made.
SMXE_INV_PAR	Invalid parameter.
SMXE_WRONG_HEAP	cp not in heap hn.

Descr

smx_HeapScan() is intended to perform frequent heap scans and to fix or report heap problems that it finds. Normally it is called once per pass of the idle task and scans fnum chunks forward or bnum chunks backward. It cannot be interrupted by another heap service while scanning.

cp can be set to start a scan at a specific chunk in the heap. However, it is usually set to NULL, in which case, the new scan starts from where the last scan left off, at hsp. Repetitively calling smx_HeapScan() with cp == NULL, results in forward scanning through the entire heap, fnum chunks at a time, until the end of the heap is reached. Then TRUE is returned to indicate that the scan is complete. When the end of the heap has not been reached, FALSE is returned to indicate to keep scanning. If smx_HeapScan() is called when the end of the heap has been reached, scanning starts from the beginning of the heap.

smx_HeapScan() fixes broken backward links by scanning forward and broken forward links by scanning backward. It also checks chunk control block (CCB) fields and fixes them, if possible. Whenever a fix is made, SMXE_HEAP_FIXED is reported.

For a debug chunk, the lower and upper fences are checked. If a broken fence is found for the debug version of smx (SMX_BT_DEBUG == 1), smx_HeapScan() reports SMXE_HEAP_FENCE_BRKN and returns TRUE. This stops the scan so that the broken fence can be inspected. In the release version, broken fences are fixed, SMXE_HEAP_FIXED is reported, and the scan continues.

If the backward scan finds a broken back link before it reaches the chunk with a broken forward link, it is not possible to fix either link. So, instead, the gap is bridged from one chunk to the other and EH_HEAP_BRKN is reported. This leaves the heap in a semi-operational mode, as long as none of the bridged chunks is accessed. This could be used to allow operation to continue in emergency mode, or for the purpose of a clean system shutdown. More frequent scanning will reduce the likelihood of double breaks, like this.

See the eheap User’s Guide, Reliability chapter for more information on heap scanning.

Notes

1. Because it is expected to run frequently, smx_HeapScan() makes no entries in EVB, other than those due to reported errors or fixes.

2. If smx_HeapScan() cannot fix a break, it reports SMX_HEAP_BRKN. This is treated as an irrecoverable error by the error manager, smx_EM(), which calls smx_EMExitHook(). The latter is the place to put heap recovery or system reboot code. See the smx User’s Guide, Error Management chapter.

Example

```
u32 heap_scan =  HEAP_SCAN_CNT;

void smx_HeapManager(void)
{
    ...
    if (--heap_scan == 0)
    {
        smx_HeapScan(NULL, 2, 100, heap0_hn);
        heap_scan = HEAP_SCAN_CNT;
    }
    ...
}
```

This example shows heap scanning in the heap manager, which is called by the idle task. smx_HeapScan() is called once per HEAP_SCAN_CNT passes through smx_HeapManager(). Starting from the beginning of the heap, it continuously scans 2 chunks forward per run, starting over when it reaches the end of the heap. If a broken forward link is found, it goes to the end of the heap and scans 100 chunks backward per run until it reaches and fixes the break, and then it resumes scanning forward. More chunks are scanned backward per run because it is important to fix a break quickly

If the heap has 200,000 chunks it will take 100,000 passes to scan the full heap. This might be too often, hence HEAP_SCAN_CNT is introduced. If slowed down to about one scan per tick, it would take 1000 seconds (about 17 minutes) to complete a pass.

smx_HeapSet

BOOLEAN smx_HeapSet (SMX_ST_PAR par, u32 val, u32 hn=0)

eheap eh_Set()

Type Mutex-protected function

Summary Sets the specified heap mode to ON or OFF.

Compl smx_HeapPeek()

Parameters

par	Parameter to set.
val	Value to set.
hn	Heap number.

Returns TRUE Parameter has been set.

FALSE Parameter has not been set due to error.

Errors SMXE_INV_PAR Invalid parameter

Descr Used to control heap modes. par is of type SMX_ST_PAR. Available parameters are:

SMX_ST_AUTO	Automatic free chunk merge control.
SMX_ST_DEBUG	Debug mode control.
SMX_ST_FILL	Block fill mode control.
SMX_ST_MERGE	Free chunk merge control.

and the available values are ON and OFF. SMX_ST_AUTO enables automatic control of chunk merge (cmerge) implemented in smx_TaskManager(). SMX_ST_DEBUG controls debug mode, which causes allocations to create debug chunks, which have additional diagnostic fields in their control blocks (see in Glossary). SMX_ST_FILL controls fill mode, which enables filling blocks with unique patterns when they are allocated or freed. It also enables filling dc and tc with unique patterns. SMX_ST_MERGE control cmerge mode, which applies to free operations. If par is not recognized, returns FALSE and reports SMX_INV_PAR.

Using this service is highly recommended over directly setting internal heap modes, which may result in incorrect settings due to preemption of the current task. Also, direct heap mode setting is not possible in umode under SecureSMX.

Example

```
smx_HeapSet(SMX_ST_MERGE, ON);
```

This example turns on cmerge mode so that blocks being freed will be merged with adjacent free blocks.

smx_HT

smx_HT

void	smx_HT_ADD (void* h, const char* name)
void	smx_HT_DELETE (void* h)
void*	smx_HTGetHandle (char* name)
const char*	smx_HTGetName (void* h)
void	smx_HTInit(void)

Types		
	smx_HT_ADD	C macro calls smx_HTAdd()
	smx_HT_DELETE	C macro calls smx_HTDelete()
	smx_HTGetHandle	Reentrant function
	smx_HTGetName	Reentrant function
	smx_HTInit()	Function

Summary Add and delete entries to the handle table (HT), query HT for handles or names, and initialize HT.

Parameters		
	h	Handle to add to the handle table or to find.
	name	Name to add to handle table or to find.

Returns none

Errors		
	SMXE_HT_DUP	Duplicate entry
	SMXE_HT_FULL	Handle table full

Descr Most smx object control blocks contain object names. HT is used to give names to objects which have no control blocks, such as ISRs and user objects.

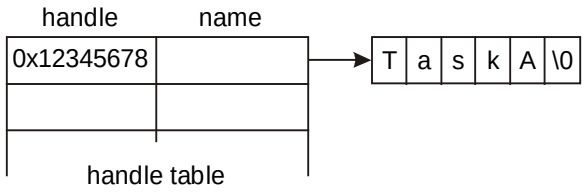
If SMX_CFG_HT (in xcfg.h), smx_HT_ADD() and smx_HT_DELETE() map onto the functions smx_HTAdd() and smx_HTDelete(), respectively. Otherwise, they map to nothing. The macros should be called instead of calling the functions directly.

smx_HT_ADD(h, name) adds an entry for handle h to HT and loads name into it. smx_HT_ADD() reports SMXE_HT_FULL if the handle table is full. If either parameter is 0, it aborts and does nothing. If SMX_CFG_HT_SCAN_DUP (in xcfg.h), smx_HT_ADD() first scans to see if name is already in the handle table. If it is, returns FALSE and reports SMXE_HT_DUP if it is.

smx_HT_DELETE(h) deletes the entry. If a name was added to HT, it must be deleted when the object is deleted.

smx_HTGetHandle() returns the handle that corresponds to the name specified, or NULL, if no entry is found. smx_HTGetName() returns the name that corresponds to the handle specified, or the null string, if no entry is found.

Handle table structure:



Example

```
void* MyISRH;
TCB_PTR  TaskA, h;
char *n;

void appl_init(void)
{
      MyISRH = smx_SysPseudoHandleCreate();
      smx_HT_ADD(MyISRH, "MyISR");
      TaskA = smx_TaskCreate(taska_main, PRI_NORM, 0, SMX_FL_NONE, "TaskA");
}

void print_report(TCB_PTR task, void *isr)
{
      const char *task_name, *isr_name;

      task_name = TaskA->name;
      isr_name = smx_HTGetName(MyISR);
      /* print report with task and ISR names */
}

void appl_exit(void)
{
      smx_TaskDelete(&TaskA);
      smx_HT_DELETE(MyISRH);
}
```

A pseudo handle is just a number that is outside the range of normal handles. See smx_SysPseudoHandleCreate(). In appl_init(), smx_HT_ADD() assigns “MyISR” to this handle and creates an entry in HT. smx_TaskCreate() loads “TaskA” into TaskA->name.

The print_report() function is able to get the task name from the task TCB and the ISR name form the handle table by using smx_HTGetName(). This enables it to print a report with names, instead of handles. smxAware uses HT in a similar way.

In appl_exit(), smx_TaskDelete() deletes the TaskA and smx_HT_DELETE() deletes the MyISR entry in HT.

smx_ISR

See the smx User's Guide, Service Routines chapter for usage information and more examples, and see the SMX Target Guide for your processor and tool suite.

smx_ISR_ENTER

smx_ISR_ENTER()

Type C and assembly macros

Summary Used to begin an smx ISR.

Compl smx_ISR_EXIT()

Parameters none

Returns none

Descr An smx interrupt service routine must begin with smx_ISR_ENTER(). Operations often performed are saving volatile registers on the current stack, switching to SS, and incrementing smx_srnest. Some processors (e.g. Cortex-M) do all of these automatically. Others require all of these to be done (e.g. some ARM's). In addition, some processors necessitate using assembly shells; others allow writing ISRs fully in C. Implementation of smx_ISR_ENTER() is a complex subject. It is discussed in detail in the SMX Target Guide.

In all cases, ISR_ENTER() saves the suspend location in task->susploc if SMX_CFG_SAVE_SUSPLOC, calls smx_RTC_ISR_START() if SMX_CFG_PROFILE, and turns Background Region on under SecureSMX.

Example 1

```
void interrupt AnISR(void)
{
    smx_ISR_ENTER();
    /* ISR body here */
    smx_ISR_EXIT();
}
```

This example is for a processor, which does hardware interrupt vectoring and which permits ISRs to be written in C. In this case, smx_ISR_ENTER() and smx_ISR_EXIT() are C macros.

Example 2

```
        EXTERN  AnISRC
        PUBLIC  AnISR
AnISR:
        smx_ISR_ENTER
        LDR     r0, =AnISRC
        MOV     lr, pc
        BX      r0
        smx_ISR_EXIT

void AnISRC(void)
{
    /* ISR body here */
}
```

This example is for a processor that does hardware vectoring, but requires assembly ISRs. ColdFire is an example. This is handled above by creating an assembly shell, AnISR, which is linked to the interrupt. It calls the ISR body, AnISRC, which written in C. It is easier to write the ISR body in C, but of course it can be written entirely in assembly, if performance is an issue. In this case, smx_ISR_ENTER() and smx_ISR_EXIT() are assembly macros.

Example 3

```
PUBLIC   smx_irq_handler

smx_irq_handler:
    smx_ISR_ENTER
    ldr     r1, =sb_IRQDispatcher
    mov     lr, pc
    bx      r1
    smx_ISR_EXIT

void AnISRC(void)
{
    /* ISR body here */
}
```

This example is for a processor that requires software vectoring. Some ARM processors are an example. This is handled by creating sb_IRQDispatcher(), which determines which ISR to call, then calls it, such as AnISRC() shown above. sb_IRQDispatcher() is supplied as part of smxBSP for the processor, and need not be written by the user. It can be found in the processor / tool assembly module (e.g. xarm_iar.s).

Normally all ISRs will be written in C for this kind of processor, since software dispatching is slow to begin with. In this case, smx_ISR_ENTER() and smx_ISR_EXIT() are assembly macros.

smx_ISR_EXIT

smx_ISR_EXIT()

Type C and assembly macros

Summary Used to end an smx ISR. Binds it to the smx scheduler.

Compl smx_ISR_ENTER()

Parameters none

Returns none

Descr All interrupt service routines which use smx_ISR_ENTER() must end with smx_ISR_EXIT().

For most processors: If smx_srnest is greater than 1, decrements smx_srnest, pops registers pushed by smx_ISR_ENTER() and does an interrupt return to the interrupted service routine or scheduler. If smx_srnest is 1, and smx_lqctr != 0, branches to the prescheduler, which calls the LSR scheduler, smx_SchedRunLSRs(), which runs all waiting LSRs. If smx_lqctr == 0, clears smx_srnest, switches to the current task stack, pops the registers pushed by smx_ISR_ENTER(), and does an interrupt return to the current task.

For ARM-M processors: smx_srnest is not needed for ISR nesting due to the RETTOBASE flag, but it is needed to check LSR plus SSR nesting. If ARMM_FL_RETTOBASE == 0, control goes to the interrupted ISR. Otherwise, if smx_srnest == 0, and smx_lqctr != 0, smx_srnest is set to 1 and the smx_PendSV_Handler() (PSVH()) is triggered. Then interrupt return is called. PSVH() tail-chains to this and runs. Since smx_lqctr != 0, PSVH() calls smx_SchedRunLSRs(), which runs all waiting LSRs.

For all processors, after all queued LSRs run, if smx_sched != 0, smx_SchedRunTasks() is called, to determine what task to dispatch next. If another task has higher priority than smx_ct, it will be dispatched unless smx_ct is locked. If smx_sched == 0, control goes to smx_ct.

If ARMM_FL_RETTOBASE == 0 and srnest != 0, control goes to the point of interruption, which could be in an LSR or in system code, such as an SSR or the smx scheduler.

Examples See smx_ISR_ENTER().

smx_LSR

See the smx User's Guide, Service Routines chapter for usage information and more examples.

smx_LSRCreate

LCB_PTR smx_LSRCreate(FUN_PTR fun, TCB_PTR htask, u32 ssz, u32 flags, const char* name, LCB_PTR* lhp=NULL)

Summary Creates an LSR.

Parameters

fun	LSR function.
htask	Host task.
ssz	LSR stack size.
flags	Flags: system, umode.
name	LSR name.
lhp	LSR handle pointer (see hp note in Notes and Restrictions).

Returns

handle	LSR created.
NULL	LSR not created due to an error.

Flags

SMX_FL_TRUST	Trusted LSR.
SMX_FL_UMODE	SecureSMX only.
SMX_FL_NOLOG	Don't log LSR in EVB.

Errors

SMXE_INV_OP	Attempted double create.
SMXE_INV_PAR	Both TRUST and UMODE flags set.
SMXE_OUT_OF_LCBS	

Descr Gets an LSR control block from the LSR control block pool and loads fun, lhp, cbtype, and name into it. If flags == SMX_FL_TRUST, sets lsr->flags.trust = 1. This is the normal LSR if SecureSMX is not in use. For SecureSMX, two additional types of LSRs are supported, called *safe LSRs*. See the SecureSMX User's Guide, section 6.7 Safe LSRs for more information.

LSR Fun void lsr_main(u32 par)

Example

```
LCB_PTR lsra

lsra = smx_LSRCreate(lsra_main, NULL, 0, SMX_FL_TRUST, "lsra", &lsra);
```

Creates lsra with function lsra_main and name "lsra". lsra is a *trusted LSR* which means that it runs in handler mode and does not have its own stack so it uses the system stack. This is the standard smx LSR.

smx_LSRDelete

BOOLEAN smx_LSRDelete(LCB_PTR* lhp)

Summary Deletes an LSR created by smx_LSRCreate().

Parameters lhp LSR handle pointer.

Returns TRUE LSR deleted.
FALSE LSR not deleted.

Errors SMXE_INV_LCB

Descr For a trusted LSR, returns the LCB back to the LCB pool, and sets *lhp = smx_nullcb so it cannot be used again. For safe LSRs, see the SecureSMX User's Guide, section 6.7 Safe LSRs for more information.

Example

```
LCB_PTR lsra

smx_LSRDelete(&lsra)
```

smx_LSRInvoke

BOOLEAN smx_LSRInvoke (LSR_PTR lsr, u32 par=0)
void smx_LSR_INVOKE (LSR_PTR lsr, u32 par=0)

Types smx_LSRInvoke SSR for use from tasks
smx_LSR_INVOKE Unrestricted function for use from ISRs and LSRs

Summary Invokes a link service routine and passes par to it.

Parameters lsr LSR to invoke.
par Parameter to pass to LSR.

Returns TRUE LSR invoked.
FALSE Error.

Errors SMXE_LQ_OVFL smx_lq is full.

Descr Places the LSR handle, lsr, followed by the parameter, par, into the LSR queue, smx_lq. If smx_LSRInvoke() is called from a task, lsr runs immediately, unless LSRs are off (see below). If smx_LSR_INVOKE() is called from an ISR or an LSR, lsr will run after all ISRs have run and all LSRs ahead of it in smx_lq have run.

LSR Main void lsr_main(u32 par)

Notes:
1. LSRs run with interrupts enabled.
2. smx_LSR_INVOKE() has no return value since there is nothing an ISR can do to retry. However, it does report SMXE_LQ_OVFL.
3. Pointer parameters: For processors with separate address and data registers, such as ColdFire, see the note about LSR and task main function parameters at the start of the Calls section.

4. smx_LSRInvoke() allows a task to invoke an LSR in the same way that an ISR invokes it, which is useful to start an interrupt-driven process. It is also useful to simulate an interrupt during debugging.

Example

```
LCB_PTR send_lsr;
SCB_PTR send_done;

void send_main(u32)
{
	MCB_PTR msg;
	char *mbp;
	u32 size;

	msg = smx_MsgGet(send_pool, &mbp, 0);
	size = smx_MsgPeek(msg, SMX_PK_SIZE);
	fill_msg(mbp, size);
	smx_LSRInvoke(send_lsr, (u32)msg);
	smx_SemTestStop(send_done, SMX_TMO_DFLT);
}

void send_next_ISR(void)
{
	smx_LSR_INVOKE(send_lsr, 0);
}

void send_lsr_main(u32 val)
{
	static char *cp;
	static MCB_PTR msg;

	switch (val) {
		case 0:
			if (*cp != '\0')
			{
				output(cp);
				cp++;
			}
			else
			(
				smx_MsgRel(msg, 0);
				smx_SemSignal(send_done);
			}
			break;
		default:
			msg = (MCB_PTR)val;
			cp = (char*)smx_MsgPeek(msg, SMX_PK_DP);
			output(cp);
			cp++;
	}
}
```

The send task gets a message, fills it, then invokes send_lsr() with the message handle as the parameter. send_lsr() loads this into the static msg, loads the first character pointer into the static cp, sends the first character, increments cp, and stops. When the output device needs the next character, it interrupts to cause send_next_ISR() to invoke send lsr() with a 0 parameter. send_lsr() sends the next character. This continues until send_lsr() reaches the null character, at which time it releases the message back to its pool and signals the send_done semaphore to send another message.

This example shows the value of being able to invoke an LSR from either a task or an ISR. In this case, invoking from a task serves to get the output process started and invoking from an ISR serves to keep it going.

smx_LSRsOff

void smx_LSRsOff (void)

Type Function

Summary Inhibits LSRs from running.

Compl smx_LSRsOn()

Descr Used in tasks to prevent LSRs from running. This makes the code atomic because an interrupt cannot cause a preemption. The effect is similar to smx_TaskLock(), except that locking does not prevent LSRs and SSRs from running.

Example

```
void atask_main(u32)
{
    smx_LSRsOff();
    atask->fun = new_function;
    smx_LSRsOn();
}
```

smx_LSRsOn

BOOLEAN smx_LSRsOn (void)

Type SSR

Summary Re-enables LSRs and runs any that are waiting

Compl smx_LSRsOff()

Returns TRUE

Errors none

Descr Re-enables LSRs. This is an SSR so that LSRs that were invoked when LSRs were off, will run before the task resumes.

Example See above.

smx_Msg

See the smx User's Guide, Exchange Messaging chapter for usage information and more examples.

smx_MsgBump

BOOLEAN smx_MsgBump (MCB_PTR msg, u8 pri)

Type SSR

Summary May change message priority; requeues the message .

Parameters

msg	Message to change priority and requeue.
pri	Priority to change to, or SMX_PRI_NOCHG.

Returns

TRUE	Success.
FALSE	Error.

Errors

SMXE_INV_MCB	Invalid message handle.
SMXE_INV_PRI	pri > SMX_MAX_PRI
SMXE_INV_XCB	Invalid exchange handle.

Descr If msg is valid and pri <= SMX_MAX_PRI, changes msg priority to pri and requeues it if it is in a valid exchange queue. If pri is SMX_PRI_NOCHG, does not change msg priority, but moves msg to the end of the queue.

Example

```
XCB_PTR  xa, xb;

void em9(void)
{
    MCB_PTR msg1, msg2;
    u8 pri2;

    msg1 = smx_MsgGet(msg_pool, NULL, 0);
    smx_MsgSend(msg1, xa);
    msg2 = smx_MsgGet(msg_pool, NULL, 0);
    smx_MsgSend(msg2, xa);
    pri2 = (u8)smx_MsgPeek(msg2, SMX_PK_PRI);
    smx_MsgBump(msg2, ++pri2);
}
```

In this example, two messages are obtained and sent to xa.. Then, smx_MsgPeek() is used to get the priority of msg2, which is bumped up by one. As a consequence, msg2 will now be ahead of msg1 in the xa message queue.

smx_MsgGet

MCB_PTR smx_MsgGet (PCB_PTR pool, u8** bpp=NULL, u16 clrsz=0, MCB_PTR* mhp=NULL)

Type SSR

Summary Gets a message by combining a message block from a block pool and an MCB from the MCB pool.

Compl smx_MsgRel()

Parameters

pool	Pool to get message block from.
bpp	Pointer to message block pointer. NULL if none.
clrsz	Number of bytes to clear from the start of message block.
mhp	Message handle pointer (see hp note in Notes and Restrictions) .

Returns

msg	Handle of message obtained.
NULL	Out of blocks or error.

Errors

SMXE_INV_PCB	Invalid pool handle.
SMXE_OUT_OF_MCBS	

Descr Gets a block from pool for use as the message block and gets an MCB from the MCB pool. Initializes the MCB, links it to the message block, clears the first clrsz bytes of the message body up to its size, and loads the address of the message block into bpp, unless bpp is NULL. bpp is intended to be used to load data into the message block. The smx_ct or smx_clsr becomes the message owner. Returns the message handle.

Notes

1. For proper operation there must be at least as many MCBs as there are active messages in a system at any given time.
2. Interrupt safe with respect to sb_BlockGet() and sb_BlockRel() operating on the same block pool.

Example

```
MCB_PTR build_msg(PCB_PTR pool, u8* dp)
{
    u8*  mbp;
    MCB_PTR  msg;

    msg = smx_MsgGet(pool, &mbp, 4);
    LoadMessage(mbp, dp);
    return msg;
}
```

This function gets a message from pool, loads data into it, and returns the message handle.

smx_MsgMake

MCB_PTR smx_MsgMake (u8* bp, u32 bs=0 , pri=0, MCB_PTR* mhp=NULL)

Type SSR

Summary Makes a message from a bare block or a protected block under SecureSMX.

Compl smx_MsgUnmake()

Parameters

bp	Block pointer.
bs	Block source: pool, heap, or no source (-1).
mhp	Message handle pointer (see hp note in Notes and Restrictions).

Returns

msg	Handle of message made.
NULL	Insufficient resources or error.

Errors

SMXE_INV_PAR	bp == NULL
SMXE_INV_OP	mhp == NULL or *mhp is invalid
SMXE_OUT_OF_MCBS	

Descr Makes a message from a bare block or a protected block. Gets MCB from MCB pool, initializes it, and returns its handle. If the block is from a pool, msg->bs = &pool; if the block is from a heap, msg->bs = hn. If the block is a standalone block, msg->bs = -1.

Example

```
LCB_PTR     in_LSR;
PCB         in_pool;
XCB_PTR     in_xchg;

void in_ISR(void);
{
    u8 char;
    u8 *mbp, *dp;

    char = UART_In();
    switch (char)
    {
        case: STX
            mbp = sb_BlockGet(&in_pool, 4);
            dp = mbp;
            break;
        case: ETX
            smx_LSR_INVOKE(in_LSR, (u32)mbp)
            break;
        default:
            *dp++ = ch;
    }
}
```

```
void in_LSR_main(u32 mbp);
{
    MCB_PTR msg;

    msg = smx_MsgMake((u8*)mbp, &in_pool);
    smx_MsgSend(msg, in_xchg);
}
```

in_ISR() runs whenever an UART input interrupt occurs. It gets an incoming character from the UART. If it is the start of text, STX, a base block is obtained from in_pool. This is an interrupt-safe function designed for ISR usage. Subsequent characters are loaded into the base block. When the end of text, ETX, is received, in_LSR is invoked. in_LSR runs after all ISRs complete. It uses smx_MsgMake() to make the base block at mbp into a message and then sends the message to in_xchg, where a task waits to process it. Note that this is a no-copy operation.

smx_MsgPeek

u32 smx_MsgPeek (MCB_PTR msg, SMX_PK_PAR par)

Type SSR

Summary Returns the current value of the parameter specified.

Parameters

msg	Message to peek at.
par	What to return.

Returns

value	Value of par.
0	Value, unless error.

Errors

SMXE_INV_MCB	Invalid message handle.
SMXE_INV_PAR	Argument not recognized.
SMXE_BROKEN_Q	Message queue is broken.
SMXE_UNKNOWN_SIZE	Message block size is unknown

Descr This service can be used to peek at a message. Valid arguments are:

SMX_PK_BP	Block pointer.
SMX_PK_HN	Heap number.
SMX_PK_ONR	Owner.
SMX_PK_NEXT	Next msg in queue. NULL, if none.
SMX_PK_PRI	Message priority.
SMX_PK_POOL	Pool.
SMX_PK_REPLY	Reply field. 0 if mcb.rpx = 0xFFFF (no reply).
SMX_PK_SIZE	Block size.
SMX_PK_XCHG	Exchange where msg is waiting. 0, if none, or broken queue.

If the message block is from a heap or a pool, SMX_PK_SIZE returns the block size. If not, 0 is returned and SMXE_UNKNOWN_SIZE is reported. Hence, if a message was made from a freestanding block, its size must be stored outside of the message.

Example

```
u8* mbp;
MCB_PTR msg;
BOOLEAN pass;
XCB_PTR xchgM, reply;

if (msg = smx_MsgReceive(xchgM, &mbp, TMO))
{
    pass = ProcessMsg(mbp);
    reply = (XCB_PTR)smx_MsgPeek(msg, SMX_PK_REPLY);
    *mbp = pass;
    smx_MsgSend(msg, reply, 0, NO_REPLY);
}
```

This is an example where a message is received from xchgM and processed. pass indicates if processing was successful. smx_MsgPeek() is used to find the reply exchange, the first byte of msg is set equal to pass, and msg is send to the reply exchange, where the sender waits for acknowledgement. Note that it is not necessary to know the origin of the message.

smx_MsgReceive

MCB_PTR smx_MsgReceive (XCB_PTR xchg, u8** bpp=NULL, u32 timeout=0, MCB_PTR* mhp=NULL)

Type SSR

Summary Gets a message from xchg. If xchg is empty, suspends the current task for timeout ticks. Fails if timeout ticks elapse before a message is received.

Compl smx_MsgSend()

Parameters

xchg	Exchange to get message from.	
bpp	Pointer to message block pointer. NULL if none.	
timeout	Timeout in ticks or msec if |SMX_FL_MSEC.	
mhp	Message handle pointer (see hp note in Notes and Restrictions).	

Returns

msg	Message handle.
NULL	Error or timeout.

Errors

SMXE_INV_XCB	Invalid exchange handle.
SMXE_INV_PRI	Message priority is invalid for a pass exchange.
SMXE_WAIT_NOT_ALLOWED	Called from LSR with nonzero timeout.

Descr If xchg is a **normal exchange**, dequeues the first message waiting at xchg and returns the message handle. The task or LSR that made the call becomes the message owner. Also loads the message body pointer into bpp for access to the message body. If xchg is empty and timeout is not 0, suspends smx_ct on xchg for timeout ticks. smx_ct is enqueued in priority order. If a message is sent to xchg before the timeout elapses, smx_ct resumes with the message handle as the return value and the message body pointer is loaded into bpp. If the timeout elapses or was 0, ct resumes with a NULL return value and nothing is loaded into bpp. Timeouts are not permitted for LSRs.

If xchg is a **pass exchange**, changes task priority if it is less than SMX_PRI_SYS. If not, returns NULL and reports SMXE_INV_PRI. If smx_ct does not own a mutex, changes smx_ct->prinorm = msg->pri and smx_ct->pri = msg->pri. If smx_ct owns a mutex, smx_ct->prinorm and smx_ct->pri are changed up, but not down, in order to preserve priority promotion, if any, by the mutex. Requeues smx_ct in the ready queue if its priority has changed. If smx_ct priority is decreased it may be preempted, unless it is locked. For an LSR, receiving from a pass exchange is the same as receiving from a normal exchange since LSRs have no priority.

If xchg is a **broadcast exchange**, and a message is waiting, smx_ct receives the message handle and the message body pointer is loaded into bpp. However, msg remains enqueued at xchg and its sender remains its owner. If no message is waiting at xchg, ct is enqueued at xchg in FIFO order. Operation for a message received before the timeout elapses or after it elapses, is similar to a normal exchange, except that msg remains enqueued at xchg and its sender remains its owner. All tasks waiting at xchg receive the message at the same time.

Notes 1. Clears smx_lockctr if called from a task and timeout != SMX_TMO_NOWAIT.

Example

```
XCB_PTR in_xchg;
MCB_PTR msg;

void task_Main(u32)
{
    u8* mbp;
    while (1)
    {
        if (msg = smx_MsgReceive(in_xchg, &mbp, 100))
            Process.Msg(mbp);
        else
            break;
    }
    /* report failure */
}
```

In the above example, task gets msg from the in_xchg and processes it, using mbp. task will wait up to 100 ticks, and if there is no message, it will report a failure.

smx_MsgReceiveStop

void smx_MsgReceiveStop (XCB_PTR xchg, u8** bpp=NULL, u32 timeout=0, MCB_PTR* mhp=NULL)

Type Limited SSR — tasks only

Summary Same as smx_MsgReceive() except that smx_ct is always stopped, then restarted when it is time for it to run.

Compl smx_MsgSend()

Parameters

xchg	Exchange to get message from.
bpp	Pointer to message block pointer. NULL if none.
timeout	Timeout in ticks or msec if |SMX_FL_MSEC.
mhp	Message handle pointer (see hp note in Notes and Restrictions).

Errors

SMXE_INV_XCB	Invalid exchange handle.
SMXE_INV_PAR	bpp points to a location in the current task stack.
SMXE_INV_PRI	Message priority is invalid for a pass exchange.
SMXE_OP_NOT_ALLOWED	Called from an LSR.

Descr See smx_MsgReceive() for operational description. smx_ct always stops, then restarts instead of suspending then resuming. The message handle is returned via the parameter in taskMain(par), when the task restarts.

Notes

1. If called from an LSR, aborts operation and returns to LSR.
2. Clears smx_lockctr if called from a task, since it always stops.

TaskMain void task_main(MCB_PTR msg)

par

handle	Message handle received.
NULL	Error or timeout.

Note: For processors with separate address and data registers, such as ColdFire, see Note 8 in smx Services, Notes and Restrictions.

Example

```
XCB_PTR input;
MCB_PTR data;
u8*  mbp;

void task_Main(u32 msg)
{
    if (msg != NULL)
    {
        ProcessData(mbp);
        smx_MsgReceiveStop(input, &mbp, TMO);
    }
    else
        /* report failure */
}
```

The above example is equivalent to the example shown for smx_MsgReceive(). Note that there is no while loop — when a message is received or a timeout occurs, smx restarts task and passes the message handle or NULL as the task_Main() parameter. Also note that mbp is defined as a static variable — it cannot be defined as an auto variable, because the stack changes.

smx_MsgRel

BOOLEAN smx_MsgRel (MCB_PTR msg, u16 clrsz=0)

Type SSR

Summary Releases a message obtained by smx_MsgGet() or smx_MsgMake().

Compl smx_MsgGet(), smx_MsgMake()

Parameters

msg	Message to release.
clrsz	Number of bytes to clear from byte 4 of message block if from a pool.

Returns

TRUE	Message released.
FALSE	Invalid MCB or msg is not owned by current task.

Errors

SMXE_INV_MCB	Invalid message handle
SMXE_INV_PAR	Message block pointer is out of range.
SMXE_NOT_MSG_ONR	smx_ct is not the message owner.
SBE_INV_POOL	Invalid pool handle.
SBE_INV_BP	Block pointer is out of range.

Descr Releases a message obtained by smx_MsgGet() or smx_MsgMake(). If msg is at a broadcast exchange, dequeues it. If msg->bs < eh_hvpn, msg is from a heap, and hn = msg->bs. Frees msg block to hn if msg->bp is in the address range of heap hn.

If msg->bs >= eh_hvpn but != -1, msg block is from a pool and pool pointer, pp = msg->bs. Releases msg block to pool at pp if it is a valid pool and if bp is in the pool range. Also clears clrsz bytes from byte 4 to the end of the msg block.

If msg->bs == -1, the block is a standalone block, and is not released.

The operation also fails if the message is not owned by the current task. This is done for safety to prevent a task that no longer owns a message from releasing it. Note: an LSR can release a message that it does not own. This is done to allow message handles to be passed to LSRs as LSR parameters.

Notes

1. This allows a broadcast task to release a message it sent to a broadcast exchange, since it still owns the message.
2. Interrupt safe with respect to sb_BlockGet() and sb_BlockRel() operating on the same block pool.

Example

```
u32  release_msgs(XCB_PTR xchg)
{
      MCB_PTR  msg;
      U32  i, sz;

      for (i = 0; (msg = smx_MsgReceive(xchg, SMX_TMO_NOWAIT)); i++)
      {
            sz = smx_MsgPeek(msg, SMX_PK_SIZE);
            smx_MsgRel(msg, sz);
      }
      return i;
}
```

All messages waiting at xchg are removed, cleared, and released. The number of messages released is returned to the caller.

smx_MsgRelAll

u32 smx_MsgRelAll (TCB_PTR task)

Type SSR

Summary Releases all messages owned by task and returns the number released.

Parameters task Task whose messages are to be released.

Returns Number of messages released.

Errors SMXE_INV_TCB Invalid task handle.

Descr Searches entire MCB pool and releases all messages owned by task. Messages are dequeued before release. Returns number of messages released.

Example

```
void stop_task(TCB_PTR atask)
{
      smx_MsgRelAll(atask);
      smx_TaskStop(atask);
}
```

Unlike smx_TaskDelete(&atask), smx_TaskStop(atask) does not automatically release all messages owned by atask. In this example, all of atask's messages are released, then it is stopped.

smx_MsgSend

BOOLEAN smx_MsgSend(MCB_PTR msg, XCB_PTR xchg, u8 pri=0, void* reply=NULL)

Type SSR

Summary Sends a message to an exchange. Delivers msg to the top waiting task, if any.

Compl smx_MsgReceive(), smx_MsgReceiveStop()

Parameters

msg	Message to send.
xchg	Exchange to send message to.
pri	Priority to set msg to unless SMX_PRI_NOCHG.
reply	Where to send reply. NULL or smx_nullcb if no reply is expected.

Returns

TRUE	Message sent.
FALSE	Message not sent due to error.

Errors

SMXE_INV_MCB	Invalid message pointer.
SMXE_NOT_MSG_ONR	smx_ct is not the owner.
SMXE_INV_XCB	Invalid exchange pointer.
SMXE_INV_PAR	Reply is not a valid exchange handle or index or pri is invalid.

Descr If xchg is a **normal exchange**, msg is enqueued in its wait queue, unless there is a task waiting. If so, msg is delivered to the first task. This task becomes the new owner and it is resumed. If there is no task waiting at the exchange, msg is enqueued in priority order, unless its priority is 0, in which case it is enqueued in FIFO order. If pri == SMX_PRI_NOCHG, the message priority is not changed. This allows a task to forward a message without changing its priority. xchg becomes the message owner so the message will not be released if the task is deleted or smx_MsgRelAll(task) is called.

If xchg is a **pass exchange**, operation is the same with the addition that the task receiving msg assumes it priority. See the discussion of this feature under smx_MsgReceive().

If xchg is a **broadcast exchange, bxchg**, operation is quite different. Tasks are enqueued in FIFO order and all are resumed at once by smx_MsgSend(). Each task receives the msg handle, and the message block pointer is loaded into its mbp location. However, the sending task remains the owner, and the message “sticks” to the bxchg, meaning that it will be “received” by all subsequent receives until replaced or released.

The broadcast message can be replaced by sending another message to the bxchg or it can be released by the initial sender. In the first case, the message will be automatically released and any task can cause this to happen. In the second case, the initial sender is the message owner, so only it can release the message. See smx User’s Guide, Exchange Messaging chapter, broadcasting messages section for more discussion of broadcasting.

Notes The following notes apply to all cases above:

1. If a task making this call is not the message owner, the call returns FALSE and reports SMXE_NOT_MSG_ONR. This prevents a task from sending a message that it does not own. However an LSR can send a message that it does not own. This allows LSRs to release messages that they did not get nor make.

2. msg, xchg, and reply are checked and if invalid, FALSE is returned and the appropriate error is reported.

3. The reply parameter allows the sender to tell the msg recipient where to reply. It is an exchange handle 8-bit index, which is stored in msg->rpx. If the reply parameter == NULL or smx_nullcb, then rpx is set to 0xFF, meaning no reply. See smx User's Guide, Exchange Messaging chapter, using the reply field section for more discussion.

4. If xchg->cbfun is not NULL, the callback function cbfun(XCB_PTR xchg) is called. See also smx_MsgXchgSet().

Example1

```
typedef struct
{
      u32  hdr;
      u8  data[N];
} *MB_PTR;

PCB_PTR   free_msgs;
XCB_PTR   port0;

BOOLEAN send_msg(void)
{
      MCB_PTR  msg;
      MB_PTR  mbp;

      if (msg = smx_MsgGet(free_msgs, &mbp, SMX_TMO_NOWAIT))
      {
            mbp->hdr = TEST;
            for (i = 0; i < N; i++)
                  mbp->data[i] = i;
            smx_MsgSend(msg, port0);
            return TRUE;
      }
      else
            return FALSE;
}
```

In this example, a message block is obtained, filled with a test pattern, and sent to another exchange called port0. Message priority is set to 0 and no reply is expected. Returns TRUE if a message is sent, FALSE otherwise.

Example 2 See smx User's Guide, Exchange Messaging chapter, client/server example for a reply example.

smx_MsgUnmake

u8* smx_MsgUnmake (MCB_PTR msg, u32* bsp=NULL)

Type SSR

Summary Unmakes a message made by smx_MsgMake() to a bare block or to a pblock.

Compl smx_MsgMake()

Parameters

msg	Message to unmake.
bsp	Place to put block source.

Returns

>0	Message unmade.
NULL	Invalid MCB or msg is not owned by current task.

Errors

SMXE_INV_MCB	Invalid message handle.
SMXE_NOT_MSG_ONR	smx_ct is not the message owner.

Descr Reverses smx_MsgMake() by converting an smx message to a bare block or pblock[5] by releasing its MCB. A task that no longer owns a message cannot unmake it. However, an LSR can unmake a message that it does not own. Returns the address of the data block and loads its source into the user-supplied location, *bsp, unless bsp == NULL.

Example

```
u8* bpi;
PCB_PTR  msg_pool;
LCB_PTR  out_LSR;
PCB_PTR  ppi;

void SendMsg(void)
{
      u8*  mbp;
      MCB_PTR  msg;

      msg = smx_MsgGet(msg_pool, &mbp, 4);
      /* load NULL terminated message using mbp */
      smx_LSRInvoke(out_LSR, (u32)msg);
}

void out_LSR_main(u32 m)
{
      MCB_PTR msg = (MCB_PTR)m;

      bpi = smx_MsgUnmake(msg, &ppi,);
      UART_Out(*bpi++);
}
```

[5] See the SecureSMX User's Guide for information on pblocks.

```
void out_ISR(void)
{
	if (*bpi != 0)
	{
		UART_Out(*bpi++);
	}
	else
	{
		sb_BlockRel(ppi, bpi, 0);
		UART_Stop();
	}
}
```

This example is the opposite of that shown for smx_MsgMake(). It is assumed that a task calls SendMsg(), which gets a message, loads it, then invokes out_LSR with msg as its parameter. out_LSR unmakes the message, thus loading bpi and ppi for out_ISR(). out_LSR then outputs the first character to start UART output. The UART interrupts each time it needs another character, and out_ISR() provides the character until all characters have been sent. out_ISR() releases the block back to msg_pool, which is pointed to by ppi.

smx_MsgXchg

See the smx User's Guide, Exchange Messaging chapter for usage information and more examples.

smx_MsgXchgClear

BOOLEAN smx_MsgXchgClear (XCB_PTR xchg)

Type SSR

Summary Clears an exchange.

Parameters

xchg	Exchange to clear.	

Returns

TRUE	Exchange cleared or already clear.
FALSE	Error.

Errors

SMXE_INV_XCB	Invalid exchange handle.
SMXE_BROKEN_Q	Task or message queue is broken.

Descr At the time it is cleared, a message exchange can have a task queue, a message queue, or no queue. Clears an exchange by resuming all waiting tasks with NULL return values, for a task queue, or releasing all waiting messages, for a message queue. Appropriate xchg fields are cleared.

Example

```
BOOLEAN  modeA;
XCB_PTR  port_in;
TCB_PTR  serverA, serverB;

BOOLEAN toggle_server(void)
{
    BOOLEAN pass = FALSE;

    smx_TaskLock();
    if (modeA)
    {
        pass  = smx_TaskStop(serverA);
        pass &= smx_MsgXchgClear(port_in);
        pass &= smx_TaskStart(serverB, port_in);
    }
    else
    {
        pass  = smx_TaskStop(serverB);
        pass &= smx_MsgXchgClear(port_in);
        pass &= smx_TaskStart(serverA, port_in);
    }
```

```
    modeA = if modeA ? FALSE : TRUE;
    smx_TaskUnlock();
    return pass;
}
```

This function toggles the task serving port_in. It does so by stopping the current server, clearing the port_in exchange, then starting the alternate server. This is done with ct locked so that the operation is atomic. Since all messages have been released, client tasks will presumably time out and try again.

smx_MsgXchgCreate

XCB_PTR smx_MsgXchgCreate (SMX_XCHG_MODE mode, const char* name=NULL, XCB_PTR* xhp=NULL)

Type SSR

Summary Creates a message exchange, which operates in the specified mode.

Compl smx_MsgXchgDelete()

Parameters

mode	Operating mode.
name	Name to give exchange, NULL if none.
xhp	Exchange handle pointer (see hp note in Notes and Restrictions).

Returns

xchg	Handle of exchange created.
NULL	Insufficient resources or error.

Errors

SMXE_INV_OP	Attempted multiple creates of the same message exchange.
SMXE_OUT_OF_XCBS	
SMXE_WRONG_MODE	Mode is not recognized.

Descr Creates an exchange of the mode specified:

mode	exchange
SMX_XCHG_NORM	Normal
SMX_XCHG_PASS	Pass
SMX_XCHG_BCST	Broadcast

Allocates an exchange control block from the XCB pool, initializes it, and returns the exchange handle.

Example

```
XCB_PTR  port_in, port_out;

void appl_init(void)
{
    port_out = smx_MsgXchgCreate(SMX_XCHG_NORM, "port_out");
    port_in  = smx_MsgXchgCreate(SMX_XCHG_PASS, "port_in");
}
```

This example shows the creation of a normal exchange and a pass exchange.

smx_MsgXchgDelete

BOOLEAN smx_MsgXchgDelete (XCB_PTR* xhp)

Type SSR

Summary Deletes an exchange created by smx_MsgXchgCreate().

Compl smx_MsgXchgCreate()

Parameters

xhp	Exchange to delete.

Returns

TRUE	Exchange deleted or already deleted.
FALSE	Error.

Errors

SMXE_INV_XCB	Invalid exchange handle.
SMXE_BROKEN_Q	Task or message queue is broken.

Descr Deletes an exchange created by smx_MsgXchgCreate(). Resumes all waiting tasks with FALSE return values or releases all waiting messages. Clears and releases the XCB, and sets *xhp = smx_nullcb so it cannot be used again.

Example

```
BOOLEAN remove_server(TCB_PTR serverA, XCB_PTR port_in)
{
    BOOLEAN pass = FALSE;

    if (smx_TaskStop(serverA))
    {
        if (smx_MsgXchgDelete(port_in))
            pass = TRUE;
    }
    return pass;
}
```

In this example, serverA is first stopped; if successful, port_in is deleted. Returns TRUE if both succeed. Normally only one server task serves a server exchange. It makes sense if it has been stopped to release all messages waiting at its exchange, since they will not be serviced. Deleting the exchange ensures that more messages cannot be sent.

smx_MsgXchgPeek

u32 smx_MsgXchgPeek (XCB_PTR xchg, SMX_PK_PAR par)

Type SSR

Summary Returns the current value of the parameter specified.

Parameters

xchg	Message exchange to peek at.
par	What to return.

Returns

value	Value of par.
0	Value, unless error.

Errors SMXE_INV_XCB Invalid exchange handle.
SMXE_INV_PAR Invalid argument.

Descr This service can be used to peek at an exchange. Valid arguments are:

SMX_PK_TASK	First task waiting on this exchange. NULL, if none.
SMX_PK_MSG	First message waiting on this exchange. NULL, if none.
SMX_PK_MODE	Mode of the exchange (BCST, PASS or NORM).
SMX_PK_NAME	Name of the exchange.

Example

```
u32 count_msgs(XCB_PTR xchg)
{
    MCB_PTR  msg;
    u32  ctr = 0;

    smx_TaskLock();
    if (msg = (MCB_PTR)smx_MsgXchgPeek(xchg, SMX_PK_MSG))
    {
        for ( ; msg->cbtype !=  SMX_CB_MCB; msg = smx_MsgPeek(cb, SMX_PK_NEXT))
            ctr++;
    }
    smx_TaskUnlock();
    return ctr;
}
```

This function returns the number of messages waiting at xchg. Note the combined use of smx_MsgXchgPeek() and smx_MsgPeek(). It is necessary to lock the current task in order to achieve accurate results.

smx_MsgXchgSet

BOOLEAN smx_MsgXchgSet(XCB_PTR xchg, SMX_ST_PAR par, u32 v1, u32 v2)

Type SSR

Summary Provides message exchange control.

Compl smx_MsgXchgPeek()

Parameters

xchg	Exchange to set.
par	Parameter to set.
vl	Value 1.
v2	Value 2.

Returns

TRUE	Parameter has been set.
FALSE	Parameter has not been set due to error.

Errors

SMXE_INV_XCB	Invalid exchange handle.
SMXE_PRIV_VIOL	Privilege violation; cannot call from umode (SecureSMX).

Descr par is of type SMX_ST_PAR. Available parameters are:

SMX_ST_CBFUN	Message send callback function = v1.

Loads the message send callback function into the exchange control block. Using this service is recommended over directly setting internal exchange modes, which may result in incorrect settings due to preemption of the current task. Direct exchange mode setting is not permitted in umode under SecureSMX.

Example

```
MCB_PTR  msg;
SCB_PTR   sema;
TCB_PTR   taska;
XCB_PTR   xchga, xchgb, xchg;

smx_MsgXchgSet(xchga, SMX_ST_CBFUN, xchga_cbfun);
smx_MsgXchgSet(xchgb, SMX_ST_CBFUN, xchgb_cbfun);

void xchga_cbfun(XCB_PTR xchg)
{
      xchg = xchga;
      smx_SemSignal(sema);
}

void xchgb_cbfun(XCB_PTR xchg)
{
      xchg = xchgb;
      smx_SemSignal(sema);
}

void taskaMain(u32 par)
{
      u8*  bp;

      while (smx_SemTest(sema, 100)
      {
            msg = smx_MsgReceive(xchg, &bp, 100);
            ProcessMsg(bp);
            smx_MsgRel(msg);
      }
}
```

This example shows how a single taska can wait for and process messages from two different exchanges, at the same time. When either xchga or xchgb receives a message, its callback function runs, which sets xchg to the appropriate xchg and signals sema, where taska waits. taska receives and processes the waiting message, then releases it.

smx_Mutex

See the smx User's Guide, Mutexes chapter for usage information and more examples.

smx_MutexClear

BOOLEAN smx_MutexClear (MUCB_PTR mtx)

Type SSR

Summary Frees mtx regardless of owner and nesting count and clears mtx task queue by resuming all tasks in it, with FALSE returns.

Compl smx_MutexGet()

Parameters

mtx	Mutex to clear.	

Returns

TRUE	Mutex cleared.
FALSE	Error.

Errors

SMXE_BROKEN_Q	Task mutex-owned queue is broken.
SMXE_INV_MUCB	Invalid mutex handle.

Descr If the mutex is owned by a task, removes the mutex from the owner's mutex-owned queue, and adjusts the priority of the owner to that of the highest priority mutex that it still owns or to normpri, if none. Requeues owner if it is in a queue and its priority has changed. Clears mtx onr and ncnt fields. Resumes all tasks waiting at mtx with FALSE returns.

Note Normally, a task should call smx_MutexRel()to release a mutex that it owns. smx_MutexClear() is used when deleting a mutex and for recovery.

Example

```
MUCB_PTR mtx;

void task_main(u32)
{
    mtx = smx_MutexCreate(1, PRI_HI, "mtx");
    /* use mtx */
    smx_MutexDelete(&mtx);
}
```

smx_MutexDelete() calls smx_MutexClear().

smx_MutexCreate

MUCB_PTR smx_MutexCreate (u8 pi, u8 ceiling, const char* name=NULL , MUCB_PTR* muhp=NULL)

Types SSR

Summary Creates a mutex.

Compl smx_MutexDelete()

Parameters

pi	Enable priority inheritance if != 0.
ceiling	Ceiling priority of mutex if != 0.
name	Name to give mutex or NULL for none.
muhp	Mutex handle pointer (see hp note in Notes and Restrictions).

Returns

handle	Mutex created.
NULL	Insufficient resources or error.

Errors SMXE_OUT_OF_MUCBS

Descr Gets mutex control block from mutex control block pool and initializes it. If pi != 0, priority inheritance is enabled. If ceiling, specifies the ceiling priority of the mutex. These are used to avoid unbounded priority inversions of tasks. If ceiling >= SMX_PRI_NUM, sets ceiling = SMX_PRI_NUM-1.

Example

```
MUCB_PTR mtx;

void task_main(u32)
{
    mtx = smx_MutexCreate(1, PRI_HI, "mtx");
    ...
    smx_MutexGet(mtx, TDFLT);
    /* protected critical section */
    smx_MutexRel(mtx);
}
```

This creates a mutex which has a ceiling at PRI_HI and priority promotion for tasks above that priority.

smx_MutexDelete

BOOLEAN smx_MutexDelete (MUCB_PTR* muhp)

Type SSR

Summary Deletes a mutex created by smx_MutexCreate().

Compl smx_MutexCreate()

Parameters

muhp	Mutex to delete.

Returns TRUE Mutex deleted.
FALSE Error.

Errors SMXE_INV_MUCB Invalid mutex handle.

Descr Clears the mtx task queue by resuming all tasks in it with FALSE, removes the mutex from the owner's mutex-owned list, and adjusts the priority of the owner to that of the highest priority mutex it still owns or to normpri, if none. Then clears the MUCB, releases it to the MUCB pool, and sets *muhp = smx_nullcb so it cannot be used again.

Example

```
MUCB_PTR mtx;

void task_main(u32)
{
    mtx = smx_MutexCreate(1, 0, "mtx");
    /* use mtx */
    smx_MutexDelete(&mtx);
}
```

smx_MutexFree

BOOLEAN smx_MutexFree (MUCB_PTR mtx)

Type SSR

Summary Frees the mutex regardless of owner and nesting count.

Compl smx_MutexGet()

Parameters mtx Mutex to free.

Returns TRUE Mutex freed.
FALSE Error.

Errors SMXE_BROKEN_Q Owner's mutex owned queue is broken. Does not abort.
SMXE_INV_MUCB Invalid mutex handle.

Descr Makes the next waiting task the new owner or frees the mutex if no other task is waiting. Resumes the owner with FALSE and adjusts its priority to its highest owned mutex priority or to normal priority, if none. The owner is requeued, if its priority changes. Removes mtx from previous owner's mutex owned queue.

Differs from smx_MutexRel() in that smx_ct does not need to be the owner, and the nesting count is ignored. Differs from smx_MutexClear() in that it does not clear the mtx task wait queue.

Normally, smx_MutexRel() is what a task should call to release a mutex it owns. smx_MutexFree() is called by smx_TaskDelete() if the task owns mtx. It also should be called before stopping a task that owns mtx.

Example

```
void stop_task(TCB_PTR task)
{
    MTX_PTR  mtx;

    while (mtx = smx_TaskPeek(task, SMX_PK_MTX))
    {
        smx_MutexFree(mtx);
    }
    smx_TaskStop(task);
}
```

This function frees all mutexes owned by task before stopping it. It can be called from any task, since the task does not need to own the mutexes.

smx_MutexGet

BOOLEAN smx_MutexGet (MUCB_PTR mtx, u32 timeout=0)

Types SSR

Summary Gets mutex, if free and returns TRUE. Otherwise, smx_ct is suspended in mutex's wait queue.

Compl smx_MutexRel(), smx_MutexFree(), smx_MutexClear()

Parameters

	mtx	Mutex to get.
	timeout	Timeout in ticks or msec if |SMX_FL_MSEC.

Returns

TRUE	Got mtx.
FALSE	Did not get mtx due to error or timeout.

Errors SMXE_INV_MUCB Invalid mutex handle.

Descr If mtx is free, smx_ct gets it and becomes its owner. The onr field of the MUCB is set to the task's handle, and the mutex is added to the task's mutex-owned list. If the mutex has a ceiling priority that is higher than the task's current priority, the task's priority is promoted to the ceiling priority. If the task already owns the mutex, the mutex's nesting counter is incremented and TRUE is returned.

If another task already owns the mutex and timeout is non-zero, smx_ct is suspended and priority enqueued in the mtx wait queue. If priority inheritance is enabled in mtx, the priority of the owner, if less, is promoted to smx_ct->pri, and the owner is requeued. This enables the owner to finish its operation without preemption by mid-priority tasks, which might cause unbounded priority inversion.

If the owner is waiting for another mutex, which also has priority promotion enabled, and its owner has lower priority than smx_ct->pri, then its owner's priority is promoted. Priority promotion can propagate through any number of mutexes.

If called from an LSR, TRUE is returned if the mutex is not owned by a task; otherwise FALSE is returned. Since an LSR has higher priority than any task, it is allowed to *borrow a free mutex*. This is necessary to allow LSRs to do heap operations and for other purposes. It is

safe, since the LSR will finish before any task can run, and also LSRs cannot preempt each other. If FALSE is returned, the mutex is owned by a task, and the LSR must not execute code protected by the mutex.

Notes 1. Clears smx_lockctr if called from a task and timeout > 0.

Example

```
MUCB_PTR mtx;

void taskMain(u32)
{
    smx_MutexGet(mtx, tmo);
    /* perform critical section */
    smx_MutexRel(mtx);
}
```

This example shows protecting a critical section of code by getting a mutex, then releasing it.

smx_MutexGetFast

BOOLEAN smx_MutexGetFast (MUCB_PTR mtx, u32 timeout=0)

Types Internal function

Summary Gets mutex, if free, and returns TRUE. Otherwise smx_ct is suspended in mutex's wait queue.

Compl smx_MutexRelFast()

Parameters

mtx	Mutex to get.
timeout	Timeout in ticks or msec if |SMX_FL_MSEC.

Returns

TRUE	Got mutex.
FALSE	Timeout.

Errors none

Descr Operation is the same as smx_MutexGet() except that ceiling priority is not implemented and no error checking is performed. This function is intended for use by heaps and in similar situations where speed is important and errors are unlikely. It is not an SSR but it is task safe and can be used outside of SSRs. Only smx_MutexRelFast() can be used to release mtx.

Example See smx_MutexGet() example.

smx_MutexGetStop

void smx_MutexGetStop (MUCB_PTR mtx, u32 timeout=0)

Type Limited SSR — tasks only

Summary Same as smx_MutexGet() except that ct is always stopped, then restarted when it is time for it to run.

Compl smx_MutexRel(), smx_MutexFree(), smx_MutexClear()

Parameters

mtx	Mutex to get.	
timeout	Timeout in ticks or msec if |SMX_FL_MSEC.	

Errors

SMXE_INV_MUCB	Invalid mutex handle.
SMXE_OP_NOT_ALLOWED	Called from an LSR.

Descr See smx_MutexGet() for operational description. smx_ct always stops instead of suspending, then restarts instead of resuming. Pass or fail is returned via the parameter in taskMain(par), when the former ct restarts.

Notes
1. If called from an LSR, returns to LSR and reports SMXE_OP_NOT_ALLOWED.
2. Clears smx_lockctr if called from a task, since it always stops.

TaskMain void task_main(BOOLEAN par)

par

TRUE	Got mutex.
FALSE	Error or timeout.

Example

```
MUCB_PTR mtx;
u32 pass = -1;

void taskA_Main(u32 pass)
{
    switch(pass)
    {
        case -1:
            /* initialize taskA */
        case 0:
            /* timeout or error */
        case 1:
            /* do critical section */
            smx_MutexRel(mtx);
    }
    smx_MutexGetStop(mtx, 100);
}
```

The above example shows protecting a critical section with a mutex for a one-shot task. When first started, since par = -1, taskA_Main does initialization, then attempts to get mtx and stops. When it gets mtx, it restarts, and since par == 1, it does the critical section. It then releases mtx, so another task can run and attempts to get mtx again and stops. taskA waits up

to 100 ticks. If it fails to get mtx, since par == 0, it does not enter the critical section, but rather recovers from the timeout or error and tries again.

smx_MutexRel

BOOLEAN smx_MutexRel (MUCB_PTR mtx)

Type SSR

Summary Releases mtx if owned by smx_ct and its nesting count == 1; if > 1 decrements count.

Compl smx_MutexGet(), smx_MutexGetStop()

Parameters

mtx	Mutex to release.	

Returns

TRUE	Mutex released.
FALSE	Error.

Errors

SMXE_INV_MUCB	
SMXE_MTX_ALRDY_FREE	
SMXE_MTX_NON_ONR_REL	smx_ct does not own mtx.
SMXE_BROKEN_Q	smx_ct mutex-owned queue is broken.

Descr Decrements mtx nesting count and if not zero, returns with TRUE. If nesting count is zero, removes mtx from the smx_ct mutex-owned list. If smx_ct does not own any other mutexes, its priority is restored to its normal priority, prinorm. Otherwise, smx_ct->pri is set to the highest ceiling priority or the highest waiting task priority for other mutexes owned by smx_ct that have a ceiling priority or priority inheritance enabled, respectively. If smx_ct's priority changes, it is requeued in smx_rq and preemption test is enabled. If one or more tasks are waiting in the mtx wait list, the top task is made the new mtx owner, mtx is put into its mutex-owned list, and task priority is adjusted, as appropriate. This is the service that normally should be called to release a mutex obtained with smx_MutexGet(). See also smx_MutexClear() and smx_MutexFree().

If called from an LSR, aborts and returns TRUE. See LSR discussion in smx_MutexGet().

Example See the smx_MutexGet() example.

smx_MutexRelFast

void smx_MutexRelFast (MUCB_PTR mtx)

Type Internal Function

Summary Releases mtx obtained by smx_MutexGetFast() and owned by smx_ct.

Compl smx_MutexGet(), smx_MutexGetStop()

Parameters

mtx	Mutex to release.

Returns None

Errors None

Descr Operation is the same as smx_MutexRel() except that ceiling priority is not implemented, and no error checking is performed. This function is intended for use by mutexes and similar situations where speed is important and errors are unlikely. It is not an SSR but it is task safe and can be used outside of SSRs. Only smx_MutexGetFast() can be used to get mtx.

Example See the smx_MutexGet() example.

smx_Pipe

See the smx User's Guide, Pipes chapter for usage information and more examples. Pipes serve both for message queues and for IO. Services for the former are SSRs; services for the latter are interrupt-safe functions.

smx_PipeClear

BOOLEAN smx_PipeClear (PICB_PTR pipe)

Type SSR

Summary Clears pipe and resumes all tasks waiting to put packets.

Parameters

pipe	Pipe handle.

Returns

TRUE	Pipe Cleared.
FALSE	Error.

Errors SMXE_INV_PICB Invalid pipe handle

Descr Resumes all tasks waiting on the pipe with FALSE and clears pipe_put and pipe_front flags in TCBs. Sets pipe read and write pointers to the start of the pipe buffer, thus clearing the pipe. Also clears full flag in PICB. Intended for use from tasks or LSRs. Is protected from interrupts.

Example

```
BOOLEAN restart_pipe_operation(PICB_PTR pipe)
{
    return(smx_PipeClear(pipe));
}
```

smx_PipeCreate

PICB_PTR smx_PipeCreate (void* ppb, u8 width, u16 length, const char* name=NULL , PICB_PTR* php=NULL)

Type SSR

Summary Creates a pipe.

Compl smx_PipeDelete()

Parameters

ppb	Pointer to pipe buffer.
width	Width of pipe in bytes. Can be 1 to 255. Pipe cell size = pipe width.
length	Length of the pipe in cells; can be up to 64K – 1.
name	Name to give pipe; NULL, if none.
php	Pipe handle pointer (see hp note in Notes and Restrictions).

Returns handle Pipe created.
NULL Pipe not created due to error.

Errors SMXE_INV_PAR ppb == NULL, width == 0, or length == 0
SMXE_OUT_OF_PICBS

Descr Gets a PICB and initializes it. Accepts the block pointed to by ppb as the pipe buffer. Loads pipe name, if any, into PICB. Returns address of PICB as the pipe handle and also loads into &php. The cell size determines the maximum packet size that the pipe will accept.

Notes 1. The pipe buffer must be >= width * length bytes. If it is larger there is no problem, but if it is smaller, then pipe data will overwrite whatever is after the pipe. To be safe, the user should allocate space for (width * length).

2. For best performance, pipe buffer should be aligned on a 32-bit or cache-line boundary and located in SRAM.

Example

```
#define PW 8
#define PL 10

u8 pbuf[PW*PL];
PICB_PTR pkt_pipe;

void pipe_init(void)
{
    pkt_pipe = smx_PipeCreate(pbuf, PW, PL, "pkt_pipe");
}
```

This example creates an 8-byte-wide packet pipe. An array is defined for the pipe buffer. Buffers can be statically defined, as shown, or obtained from a block pool or a heap. It is recommended to use constants for width and length. If, for example, PL were changed to 20, the pipe buffer would automatically be re-sized.

smx_PipeDelete

void* smx_PipeDelete (PICB_PTR* php)

Type SSR

Summary Deletes a pipe.

Compl smx_PipeCreate()

Parameters php Pipe handle pointer.

Returns pbuf Pipe buffer address.
0 Pipe not deleted due to error.

Errors SMXE_INV_PICB Invalid pipe handle.

Descr Deletes a pipe by resuming all waiting tasks with FALSE return values, releasing its PICB back to the PICB pool, and setting *php = smx_nullcb so it cannot be used again. Returns address of pipe buffer so the user can re-use it or release it back to its block pool or heap.

Example

```
#define PW 8
#define PL 10

PICB_PTR open_pipe(const char *name)
{
    void *ppb;

    ppb = smx_HeapMalloc(PW*PL);
    return(smx_PipeCreate(ppb, PW, PL, name));
}

BOOLEAN close_pipe(PICB_PTR pipe)
{
    void *ppb;

    ppb = smx_PipeDelete(&pipe);
    return(smx_HeapFree(ppb));
}
```

The open_pipe function shows allocating a pipe buffer from the heap using predefined width and length constants, then creating the pipe and returning its handle. The close_pipe function shows the inverse action of deleting the pipe, then using the pipe buffer address to free the buffer back to the heap. Note, in close_pipe(), that if pipe delete failed, ppb would be 0, heap free would fail, and close pipe would return FALSE.

smx_PipeGet8

BOOLEAN smx_PipeGet8 (PICB_PTR pipe, u8* bp)

Type Bare function

Summary Gets the next byte from pipe and loads it into the byte pointed to by bp. For ISR and LSR usage. Does not wake up a waiting task.

Compl smx_PipePut functions and SSRs.

Parameters

pipe	Pipe handle. Assumed to be valid.
bp	Buffer pointer to load byte.

Returns

TRUE	Byte transferred.
FALSE	Byte not transferred or pipe empty.

Errors None

Descr Gets the oldest byte in pipe, advances the pipe's read pointer to the next cell, and returns TRUE. This is the fast version of smx_PipeGet() for byte gets; it may be used in time-critical sections of user code such as in ISRs and LSRs. If this function is used in a task, it must be protected from preemption, since it is not an SSR. This function will not interfere with an interrupted complementary function that is operating on the same pipe.

Notes

1. Use only with complementary functions at the other end of the pipe.
2. Will not resume a task waiting on pipe to put a byte. Use smx_PipeResume(pipe) for that.
3. Two ISRs should not get from the same pipe.

Example

```
PICB_PTR out_pipe;

void out_chars(u8* out_port)
{
    u8 ch;

    while (smx_PipeGet8(out_pipe, &ch))
    {
        out_port = ch;
    }
}
```

In this example, all of the characters in out_pipe are sent to out_port each time the out_chars function is called. The function stops running when the pipe has been emptied.

smx_PipeGet8M

u32 smx_PipeGet8M (PICB_PTR pipe, u8* bp, u32 lim)

Type Bare function

Summary Gets the next bytes from pipe up to lim or until pipe is empty and loads them into the buffer at bp. For ISR and LSR usage. Does not wake up a waiting task.

Compl smx_PipePut functions and SSRs.

Parameters

pipe	Pipe handle. Assumed to be valid.
bp	Buffer pointer to load bytes.
lim	Limit on bytes transferred.

Returns Number of bytes transferred.

Errors None

Descr Transfers the oldest bytes in pipe to the buffer at bp, up to the limit specified or until pipe is empty, advances the pipe's read pointer and bp for each byte transferred, and returns the number of bytes actually transferred. This is faster than smx_PipeGet8() for multi-byte transfers, such as may occur with UARTs and other high-speed serial controllers that have internal buffers. It may be used in time-critical sections of user code such as in ISRs and LSRs. If this function is used in tasks, it must be protected from preemption, since it is not an SSR. This function will not interfere with an interrupted complementary function that is operating on the same pipe.

Notes

1. Use only with complementary functions at the other end of the pipe.
2. Will not resume a task waiting on pipe to put a byte. Use smx_PipeResume(pipe) for that.
3. Two ISRs should not get from the same pipe.

Example

```
PICB_PTR out_pipe;
u8  bp[10];
u32 numx;

numx = smx_PipeGet8M(out_pipe, bp, 10);
```

In this example, up to 10 bytes in out_pipe are transferred to bp[]. The limit prevents overflowing bp[]. numx is the actual number of bytes transferred.

smx_PipeGetPkt

BOOLEAN smx_PipeGetPkt (PICB_PTR pipe, void* pdst)

Type Bare function

Summary Gets the next packet from pipe and loads it into the buffer at pdst. For ISR, LSR, and task usage.

Compl smx_PipePutPkt().

Parameters

pipe	Pipe handle.
pdst	Destination pointer to store packet.

Returns

TRUE	Packet transferred.
FALSE	Packet not transferred.

Errors None

Descr If pipe is not empty, smx_PipeGetPkt() copies the oldest packet from it to the buffer at pdst, advances the pipe's read pointer to the next cell, and returns TRUE. Returns false if pipe is empty or for invalid parameter. Provides fast packet transfers. Intended primarily for use in ISRs and LSRs and is interrupt-safe. When used in tasks, it must be protected from preemption, since it is not an SSR.

Notes

1. Use only with complementary functions at the other end of the pipe.
2. Will not resume a task waiting on pipe to put a packet.
3. Two ISRs should not get from the same pipe.
4. A packet pipe (i.e. width > 1) is empty unless a full packet is present.

Example

```
PICB_PTR   pkt_pipe;
TCB_PTR    out_pkt;

ppb = smx_HeapMalloc(8*10);
pkt_pipe = smx_PipeCreate (ppb, 8, 10, “pkt_pipe”)

void out_pkt_main(u8* out_port)
{
      u8  mb[8];
      u32  i;

      while (smx_PipeGetPkt(pkt_pipe, &mb))
      {
            for (i = 0; i < 8; i++)
            {
                  *out_port = mb[i];
            }
      }
}
```

In this example, the oldest 8-byte packet is taken from the 8-byte-wide pkt_pipe. The packet is sent to out_port, byte by byte. This process continues until pkt_pipe is empty. Then out_pkt task autostops.

smx_PipeGetPktWait

BOOLEAN smx_PipeGetPktWait (PICB_PTR pipe, void* pdst, u32 timeout=0)

Type SSR

Summary Gets the next packet from pipe and loads it into the buffer at pdst. Waits if pipe is empty.

Compl smx_PipePut functions and SSRs.

Parameters

pipe	Pipe handle.	
pdst	Destination pointer to store packet.	
timeout	Timeout in ticks or msec if |SMX_FL_MSEC.	

Returns

TRUE	Packet transferred.
FALSE	Packet not transferred.

Errors

SMXE_OP_NOT_ALLOWED	Called from an LSR with timeout > 0.
SMXE_INV_PAR	pdst is NULL
SMXE_INV_PICB	Invalid pipe handle

Descr If pipe is not empty, transfers the oldest packet in pipe to the buffer at pdst and advances the pipe’s read pointer to the next cell in the pipe. If another task was waiting to put a packet, puts its packet into pipe and resumes it with TRUE. If pipe is empty and timeout > 0, smx_ct is suspended until either it gets a packet or a timeout occurs. Can be used from a task or an LSR; from an LSR, timeout must be 0.

If flags.pipe_front is set for the waiting task, its packet is passed directly to smx_ct, then the waiting task is resumed with TRUE.

Notes

1. Use only with complementary functions at the other end of the pipe.
2. May be mixed with smx_PipeGetPktWaitStop() at the same end of the pipe.
3. Multiple waiting tasks are enqueued in priority order.
4. A packet pipe (i.e. width > 1) is considered empty unless a full packet is present.
5. Clears smx_lockctr if called from a task and timeout != SMX_TMO_NOWAIT.

Example

```
PICB_PTR ctrl_pipe;
PICB_PTR data_pipe;
PICB_PTR in_pipe;

TCB_PTR  inpipe_load;
TCB_PTR  pipe_fwd;

u32 ctrl_pipe_ctr;
u32 data_pipe_ctr;

typedef struct
{
      u8  dest;
      u8  data[3];
}  PIPE_MSG;

void pipe_init(void)
{
      void*  pbp;

      pbp = smx_HeapMalloc(4*10);
      ctrl_pipe = smx_PipeCreate(pbp, 4, 10, "ctrl_pipe");
      smx_PipeSet(ctrl_pipe, SMX_ST_CBFUN, (u32)pipe_cbf);

      pbp = smx_HeapMalloc(4*10);
      data_pipe = smx_PipeCreate(pbp, 4, 10, "data_pipe");
      smx_PipeSet(data_pipe, SMX_ST_CBFUN, (u32)pipe_cbf);

      pbp = smx_HeapMalloc(4*10);
      in_pipe = smx_PipeCreate(pbp, 4, 10, "in_pipe");
}

void inpipe_load_main(void)
{
      PIPE_MSG  msg_ctrl = {1, 2, 3, 4};
      PIPE_MSG  msg_data = {0, 5, 6, 7};

      smx_PipePutPktWait(in_pipe, &msg_ctrl);
      smx_PipePutPktWait(in_pipe, &msg_data);
}
```

```
void pipe_fwd_main(void)
{
    PIPE_MSG  msg_out;

    while (smx_PipeGetPktWait(in_pipe, &msg_out))
    {
        if (msg_out.dest == 1)
        {
            smx_PipePutPktWait(ctrl_pipe, &msg_out);
        }
        else
        {
            smx_PipePutPktWait(data_pipe, &msg_out);
        }
    }
}

void pipe_cbf(PICB_PTR pipe)
{
    if (pipe == ctrl_pipe)
        ctrl_pipe_ctr++;
    else if (pipe == data_pipe)
        data_pipe_ctr++;
}
```

pipe_init() creates ctrl_pipe and data_pipe, both with the pipe_cbf callback function. Then it creates in_pipe without a callback function. The inpipe_load task loads two canned messages into in_pipe. The pipe_fwd task gets the messages and distributes them to the ctrl_pipe and to the data_pipe based upon their dest fields. pipe_cbf is called each time, and it updates the appropriate counter.

smx_PipeGetPktWaitStop

void smx_PipeGetPktWaitStop (PICB_PTR pipe, void* pdst, u32 timeout=0)

Type Limited SSR — task only

Summary Same as smx_PipeGetPktWait() except that the current task is always stopped.

Compl smx_PipePut functions and SSRs.

Parameters

pipe	Pipe handle.
pdst	Destination pointer to store packet.
timeout	Timeout in ticks or msec if |SMX_FL_MSEC.

Errors

SMXE_OP_NOT_ALLOWED	Called from an LSR.
SMXE_INV_PAR	pdst is NULL
SMXE_INV_PICB	Invalid pipe handle

Descr See smx_PipeGetPktWait() for operational description. The current task is always stopped instead of suspended, then restarted instead of resumed when it is time to run. Pass or fail is returned via the parameter in taskMain(par), when task restarts.

TaskMain void task_main(BOOLEAN par)

par
TRUE Packet transferred
FALSE Packet not transferred

Notes
1. Use only with complementary functions at the other end of the pipe.
2. May be mixed with smx_PipeGetPktWait()'s at the same end of the pipe.
3. Multiple waiting tasks are enqueued in priority order.
4. A packet pipe (i.e. width > 1) is considered empty unless a full packet is present.
5. If called from an LSR, aborts operation and returns to LSR.
6. Clears smx_lockctr if called from a task, since it always stops.

Example

```
LCB_PTR  key_LSR;
PICB_PTR key_pipe;
TCB_PTR  key_task;
u8  key_buf[4];

void key_task_init(u32)
{
      void* pbp;

      pbp = smx_HeapMalloc(4*10);
      key_pipe = smx_PipeCreate(pbp, 4, 10, "key_pipe");
      smx_TaskSet(key_task, SMX_ST_FUN, key_task_main);
      smx_PipeGetPktWaitStop(key_pipe, key_buf, 100);  /* wait for first packet */
}

void key_task_main(u32)
{
      ProcessPkt(key_buf);
      smx_PipeGetPktWaitStop(key_pipe, key_buf, 100);  /* wait for next packet */
}

void key_LSR_main(u32)
{
      smx_PipeResume(key_pipe);  /* start key_task */
}
```

```
void key_ISR(void)
{
    u8 ch;
    static cc = 0;

    ch = input_key(key_port);
    smx_PipePut8(key_pipe, ch);
    if (++cc == 4)
    {
        smx_LSR_INVOKE(key_LSR, 0)  /* packet received, invoke LSR */
        cc = 0;
    }
}
```

key_task is a one shot task started with key_task_init() as its main function. key_task_init() creates key_pipe, changes the key_task main function to key_task_main and calls smx_PipeGetPktWaitStop(). key_pipe is a 4-byte wide pipe. key_ISR() accepts one byte at a time from key_port and loads the byte into key_pipe. When 4 bytes have been loaded, the packet is complete and key_LSR is invoked. It calls smx_PipeResume() to restart key_task, with the received packet in key_buf[].

Note that it is possible that several key interrupts could occur before key_task is able to run. It does no harm to call smx_PipeResume() if key_task has already been resumed and is waiting in the ready queue. As a result there could be several packets waiting in key_pipe when key_task does start running. This is ok because the get operation will immediately restart key_task for each key that it finds in key_pipe. key_task does not actually stop, it just keeps restarting, which takes no more time than resuming.

smx_PipePeek

u32 smx_PipePeek (PICB_PTR pipe, SMX_PK_PAR par)

Type SSR

Summary Returns the current value of the parameter specified.

Parameters

pipe	Pipe handle.
par	What to return.

Returns

value	Value of argument.
0	Value, unless error.

Errors SMXE_INV_PICB Invalid pipe handle.

Descr This service can be used to peek at a pipe. Valid arguments are:

SMX_PK_FULL	Pipe is full.
SMX_PK_WIDTH	Pipe width.
SMX_PK_LENGTH	Pipe length (number of cells).
SMX_PK_NUMPKTS	Number of packets in pipe.
SMX_PK_NUMTASKS	Number of tasks waiting on pipe.

Example 1

```
TCB_PTR pipe_input_task;

void regulate_pipe(PICB_PTR pipe)
{
    if (smx_PipePeek(pipe, SMX_PK_NUMPKTS) > 3)
        pipe_input_task->pri++; /* increase task priority */
    if (smx_PipePeek(pipe, SMX_PK_NUMPKTS) < 2)
        pipe_input_task->pri--; /* decrease task priority */
}
```

In this example, the number of packets in pipe is compared to 3 to increase the priority of pipe_input_task or compared to 2 to decrease it.

Example 2

```
void send_msg(const char*);

void increase_msgs(PICB_PTR pipe)
{
    if ((smx_PipePeek(pipe, SMX_PK_NUMTASKS) > 1) && (pipe->fl->flags.pipe_put == 0))
        send_msg("Increase message input rate");
}
```

In this example, if more than one task is waiting for packets and they are not waiting to put packets, a message is sent to the operator to increase the message input rate.

smx_PipePut8

BOOLEAN smx_PipePut8 (PICB_PTR pipe, u8 byte)

Type Bare function

Summary Puts byte into pipe. For ISR and LSR usage.

Compl smx_PipeGet functions and SSRs.

Parameters

pipe	Pipe handle. Assumed to be valid.
byte	Byte to put into pipe.

Returns

TRUE	Byte put into pipe.
FALSE	Byte not put into pipe.

Errors None

Descr If pipe is not full, puts byte into pipe, and advances the pipe's write pointer to the next cell, cyclically. It may be used in time-critical sections of user code such as ISRs and LSRs. If used in a task, it must be protected from preemption, since it is not an SSR. This function will not interfere with an interrupted complementary function that is operating on the same pipe.

Notes

1. Use only with complementary functions at the other end of the pipe.
2. Will not resume a task waiting on pipe to get a byte.
3. Two ISRs should not put to the same pipe.

Example

```
PICB_PTR key_pipe;  /* byte wide pipe */
u8  input_key(u8 key_port);

void key_ISR(void)
{
      u8 ch;

      ch = input_key(key_port);
      smx_PipePut8(key_pipe, ch);
      smx_LSR_INVOKE(key_LSR, 0)  /* start task via LSR */
}
```

In this example, key_ISR runs due to an interrupt when a key is available for input. It gets the key from key_port and puts it into key_pipe. It then invokes key_LSR to start the task waiting on key_pipe to process the key.

smx_PipePut8M

u32 smx_PipePut8M (PICB_PTR pipe, u8* bp, u32 lim)

Type Bare function

Summary Puts multiple bytes from buffer at bp into pipe up to lim or until pipe is full. For ISR and LSR use.

Compl smx_PipeGet functions and SSRs.

Parameters

pipe	Pipe handle. Assumed to be valid.
bp	Buffer pointer to get bytes.
lim	Limit on bytes transferred.

Returns Number of bytes transferred.

Errors None

Descr Transfers bytes from the buffer at bp to pipe, up to the limit specified or until pipe is full. Advances the pipe's write pointer and bp for each byte transferred and returns the number of bytes actually transferred. This is faster than smx_PipePut8() for multi-byte transfers. It may be used in time-critical sections of user code such as ISRs and LSRs. If this function is used in tasks, it must be protected from preemption, since it is not an SSR. smx_PipePut8M(), in an ISR, will not interfere with an interrupted complementary function in a task or LSR that is operating on the same pipe.

Notes

1. Use only with complementary functions at the other end of the pipe.
2. Will not resume a task waiting on pipe to get a byte.
3. Two ISRs should not put to the same pipe.

Example

```
PICB_PTR out_pipe;
u8  out_buf[NUM];
u32 numx;

numx = smx_PipePut8M(out_pipe, out_buf, NUM);
```

In this example, up to NUM bytes are transferred from out_buf[] to out_pipe. The limit prevents exceeding out_buf[] size. numx is the actual number of bytes transferred.

smx_PipePutPkt

BOOLEAN smx_PipePutPkt (PICB_PTR pipe, void* psrc)

Type Bare function

Summary Puts the packet from the buffer at psrc into pipe.

Compl smx_PipeGetPkt().

Parameters

	pipe	Pipe handle.
	psrc	Pointer to source of packet.

Returns

TRUE	Packet transferred.
FALSE	Packet not transferred.

Errors None

Descr If the pipe is not full, smx_PipePutPkt() copies the packet in the buffer at psrc into it, advances the pipe's write pointer to the next cell, and returns TRUE. If pipe is full does not wait and returns NULL. Provides fast packet transfers. Intended primarily for use in ISRs and LSRs and is interrupt safe. When used in tasks, it must be protected from preemption, since it is not an SSR.

Notes

1. Use only with complementary functions at the other end of the pipe.
2. Will not resume a task waiting on pipe to get a packet.
3. Two ISRs should not put to the same pipe.

Example

```
u8  in_port;
PICB_PTR  msg_pipe;  /* width = 10 */
u8  mb[10];

void input(u8 ch, u8 port);
```

```
void in_pkt_ISR(void)
{
      u32  i;

      smx_ISR_ENTER();
      for(i = 0; i < 10; i++)
            input(mb[i], in_port);
      smx_PipePutPkt(msg_pipe, &mb);
      smx_ISR_EXIT();
}
```

In this example, a 10-byte packet is being received through the serial in_port, for each interrupt. Each assembled packet is then being put into the msg_pipe, which is 10 bytes wide. These packets are probably formatted messages, having a defined structure. Hence, it makes sense for the task unloading msg_pipe to deal with a packet stream, instead of a byte stream.

smx_PipePutPktWait

BOOLEAN smx_PipePutPktWait (PICB_PTR pipe, void* psrc, u32 timeout=0, u32 mode=0)

Type SSR

Summary Puts the packet from the buffer at psrc into pipe. Waits if pipe is full..

Compl smx_PipeGetPkt functions and SSRs.

Parameters

pipe	Pipe handle.
psrc	Pointer to source of packet.
timeout	Timeout in ticks or msec if |SMX_FL_MSEC.
mode	0: Put to back of pipe, 1: Put to front of pipe.

Returns

TRUE	Packet transferred.
FALSE	Packet not transferred.

Errors

SMXE_INV_PAR	psrc is NULL.
SMXE_INV_PICB	Invalid pipe handle.
SMXE_WAIT_NOT_ALLOWED	Called from an LSR with timeout > 0.

Descr For mode == 0: If the pipe is empty and another task is waiting to get a packet, gives the packet at psrc to the waiting task, resumes it with TRUE, and returns TRUE. Else, if the pipe is not full, copies the packet at psrc into pipe at pipe's write pointer, advances write pointer to the next cell, cyclically, and returns TRUE. If the pipe is full and timeout > 0, sets smx_ct-> flags.pipe_put = 1 and smx_ct-> flags.pipe_front = 0, and suspends the smx_ct on pipe. If no timeout returns FALSE.

For mode == 1: If the pipe is empty and another task is waiting to get a packet, gives the packet at psrc to the waiting task, resumes it with TRUE, and returns TRUE. Else, if another task is not waiting and the pipe is not full, moves pipe's read pointer back one cell, cyclically, copies the packet at psrc into pipe at the read pointer, and returns TRUE. If the pipe is full

and timeout > 0, sets smx_ct-> flags.pipe_put = 1 and smx_ct-> flags.pipe_front = 1, and suspends the smx_ct on pipe. If no timeout returns FALSE.

If pipe->cbfun is not NULL, the callback function cbfun(PICB_PTR pipe) is called. See also smx_PipeSet().

Notes

1. Use only with complementary functions at the other end of the pipe.
2. May be mixed with smx_PipePutPktWaitStop()'s at the same end of the pipe.
3. Multiple waiting tasks are enqueued in priority order.
4. Clears smx_lockctr if called from a task and timeout is not 0.

Example See smx_PipeGetPktWait() example. As shown in the example, because pipe_cbf() accepts a pipe handle as a parameter, it can be shared between pipes in a system to record how many times each pipe msg is put.

smx_PipePutPktWaitStop

void smx_PipePutPktWaitStop (PICB_PTR pipe, void* psrc, u32 timeout=0, u32 mode=0)

Type Limited SSR — tasks only

Summary Same as smx_PipePutPktWait() except that ct is always stopped.

Compl smx_PipeGetPkt functions and SSRs.

Parameters

pipe	Pipe handle.
psrc	Pointer to source of packet.
timeout	Timeout in ticks or msec if |SMX_FL_MSEC.
mode	0: Put to back of pipe, 1: Put to front of pipe.

Errors

SMXE_INV_PAR	psrc is NULL.
SMXE_INV_PICB	Invalid pipe handle.
SMXE_OP_NOT_ALLOWED	Called from an LSR.

Descr See smx_PipePutPktWait() for operational description. The current task always stops, instead of suspending, then restarts instead of resuming. Pass or fail is returned via the parameter in taskMain(par), when task restarts. If pipe->cbfun is not NULL, the callback function cbfun(PICB_PTR pipe) is called. See also smx_PipeSet().

TaskMain void task_main(BOOLEAN par)

par

TRUE	Packet transferred.
FALSE	Packet not transferred.

Notes

1. Use only with complementary functions at the other end of the pipe.
2. May be mixed with smx_PipePutPktWait()'s at the same end of the pipe.
3. Multiple waiting tasks are enqueued in priority order.
4. If called from an LSR, aborts operation and returns to LSR.
5. Clears smx_lockctr if called from a task, since it always stops.

Example

```
PICB_PTR crt_pipe;
TCB_PTR  crt_task;
u8       crt_buf1[8];
u8       crt_buf2[8];
u8       pkt_ctr = 0;

void sys_init(void)
{
       crt_task = smx_TaskCreate(crt_task_init, TP2, 0, 0, "crt_task");
       smx_TaskStart(crt_task);
}

void crt_task_init(u32)
{
       void* pbp;

       pbp = smx_HeapMalloc(8*4);
       crt_pipe = smx_PipeCreate(pbp, 8, 4, "crt_pipe");
       smx_PipeSet(crt_pipe, SMX_ST_CBFUN, (u32)pipe_ctrl);
       smx_TaskStartNew (crt_task, TRUE, SMX_PRI_NOCHG, crt_task_main);
}

void crt_task_main(u32)
{
       u8* mp;

       mp = (mp == crt_buf1 ? crt_buf2 : crt_buf1);
       smx_PipePutPktWaitStop(crt_pipe, &mp, 100);
}

void pipe_ctrl(u32 pipe)
{
       if (pipe == (u32)crt_pipe)
       {
              pkt_ctr++;
       }
}
```

In this example, sys_init() creates the one-shot crt_task with main function crt_task_init() and starts it. crt_task_init() creates crt_pipe and sets its callback function to pipe_ctrl(). crt_task then restarts itself with crt_task_main(). crt_task toggles between crt_buf1 and crt_buf2 putting an 8-byte packet from each into crt_pipe. On each put, pipe_ctrl() is called, which increments pkt_ctr.

smx_PipeResume

BOOLEAN smx_PipeResume (PICB_PTR pipe)

Type SSR

Summary Resumes first task waiting on pipe, if wait condition true.

Parameters pipe Pipe handle.

Returns TRUE Operation performed.
FALSE Operation not performed.

Errors SMXE_INV_PICB Invalid pipe handle.

Descr For the first waiting task, completes its put or get operation, if possible, and resumes the waiting task with TRUE. If put or get operation cannot be completed leaves task in the pipe wait queue and returns FALSE. Does not do put-to-front operation. If there is no task waiting, then smx_PipeResume() does nothing and returns FALSE.

An ISR can invoke an LSR to call this function in order to wake up a task waiting on pipe to put or get packets. This enables IO pipe functions at the ISR end of a pipe and pkt operations at the task end of a pipe.

Note 1. A packet pipe (i.e. width > 1) is considered empty unless a full packet is present.

Example

```
LCB_PTR  key_LSR      /* key LSR */
PICB_PTR key_pipe     /* pipe: width = 20, length = 4 */
TCB_PTR  key_task     /* key processing task */
u32  key_port;        /* serial IO port for key inputs */
u8   cc = 0;          /* input character counter */
u8   pkt[20];         /* received packet */

void key_ISR(void)
{
    u8 ch = input_key(key_port);
    smx_PipePut8(key_pipe, ch);
    if (cc++ == 20)
    {
        smx_LSR_INVOKE(key_LSR, 0);
        cc = 0
    }
}

void key_LSR_main(void)
{
    smx_PipeResume(key_pipe);
}
```

```
void key_task_main(u32)
{
    while (smx_PipeGetPktWait(key_pipe, &pkt, 100)
    {
        ProcessPkt(pkt);
    }
}
```

In this example, key task waits on key_pipe for 20-byte packets to process. The packets come in via the serial port, key_port. Each byte received by key_port causes an interrupt serviced by key_ISR. key_ISR loads each byte into the current write packet of key_pipe and counts characters as received in cc. When the cc reaches 20, a full packet has been received, and key_LSR is invoked. It resumes key_task, if it is waiting on key_pipe. Should key_task be busy processing the previous packet, nothing happens. When key_task finishes processing the last packet and returns to key_pipe, it will find the next packet or packets waiting for it and process them.

smx_PipeSet

BOOLEAN smx_PipeSet(PICB_PTR pipe, SMX_ST_PAR par, u32 v1, u32 v2)

Type SSR

Summary Provides pipe control.

Compl smx_PipePeek()

Parameters

sem	Pipe to set.
par	Parameter to set.
vl	Value 1.
v2	Value 2.

Returns

TRUE	Parameter has been set.
FALSE	Parameter has not been set due to error.

Errors

SMXE_INV_PICB	Invalid pipe handle.
SMXE_INV_PAR	par not recognized.
SMXE_PRIV_VIOL	Privilege violation; cannot call from umode (SecureSMX).

Descr par is of type SMX_ST_PAR. Available parameters are:

SMX_ST_CBFUN	Pipe put callback function = v1.

Loads the put callback function into the pipe control block. Using this service is highly recommended over directly setting internal pipe modes, which may result in incorrect settings due to preemption of the current task. Also, direct pipe mode setting is not possible in umode under SecureSMX.

Example

```
PICB_PTR      pipea;

void pipea_cbfun(PICB_PTR pipe);

smx_PipeSet(pipea, SMX_ST_CBFUN, pipea_cbfun);
```

This example loads pipea_cbfun() into the pipea control block. See smx_PipePutPktWait() for an example of usage.

smx_Sem

See the smx User's Guide, Semaphores chapter for usage information and more examples.

smx_SemClear

BOOLEAN smx_SemClear (SCB_PTR sem)

Type SSR

Summary Clears a semaphore.

Compl None

Parameters

sem	Semaphore to clear.

Returns

TRUE	Semaphore cleared.
FALSE	Semaphore not cleared due to error.

Errors SMXE_INV_SCB Invalid semaphore handle.

Descr Resumes all tasks waiting at sem with FALSE return values and deactivates their timeouts. Then resets a resource semaphore count to its original value, when created. This call would normally be used in a recovery situation, such as following a SEM_CTR_OVFL error.

Example

```
SCB_PTR printer_avail;

smx_SemClear(&printer_avail);
```

smx_SemCreate

SCB_PTR smx_SemCreate (SMX_SEM_MODE mode, u8 lim, const char* name=NULL , SCB_PTR* shp=NULL)

Type SSR

Summary Creates a semaphore of the specified mode and limit and sets its internal count, accordingly.

Compl smx_SemDelete()

Parameters

mode	Mode of operation (see below).
lim	Count limit.
name	Name to give semaphore or NULL if none.
shp	Semaphore handle pointer (see hp note in Notes and Restrictions).

Returns

handle	Semaphore created.
NULL	Semaphore not created due to insufficient resources or error.

Errors

SMXE_INV_PAR	mode or lim not in range
SMXE_OUT_OF_SCBS	

Descr Gets a semaphore control block (SCB) from the SCB pool and loads the cbtype, mode, count, lim, and name fields. Returns the address of the SCB as the semaphore handle. A semaphore is capable of operating in one of 6 modes:

mode	lim	semaphore
SMX_SEM_RSRC	1	Binary resource
SMX_SEM_RSRC	>1	Multiple resource (counting semaphore)
SMX_SEM_EVENT	1	Binary event
SMX_SEM_EVENT	0	Multiple event
SMX_SEM_THRES	t	Threshold
SMX_SEM_GATE	1	Gate

For more discussion of modes of operation, see smx User's Guide, Semaphores chapter. SMX_SEM_MODE is defined in xdef.h as an enum, for debugging convenience. If mode is not a recognized value, if lim == 0 for RSRC or THRES mode, or if lim != 1 for GATE mode, an SMXE_INV_PAR error is reported and create fails. The internal count is set to lim for RSRC semaphores and to 0 for all others.

Example

```
SCB_PTR  all_data_here, printer_avail, multi_event_sem, binary_sem;

void appl_init(void)
{
    printer_avail = smx_SemCreate(SMX_SEM_RSRC, 1, "printer_avail");
    all_data_here = smx_SemCreate(SMX_SEM_THRES, 4, "all_data_here");
    multi_event_sem = smx_SemCreate(SMX_SEM_EVENT, 0, "multi_event_sem");
    binary_sem = smx_SemCreate(SMX_SEM_EVENT, 1, "binary_sem");
}
```

appl_init() creates four semaphores: printer_avail is a binary resource semaphore, which regulates access to one printer. When a task is done with the printer it signals printer_avail. This resumes the top task waiting at printer_avail. all_data_here is a threshold semaphore, with a threshold of 4. It requires 4 signals before resuming the first waiting task. This semaphore might regulate a processing task that requires four sets of data before starting. multi_event_sem is a multiple event semaphore. It stores every event received. binary_sem is a binary event semaphore. It does not accumulate more than one event. This is useful when the task using the semaphore will process all waiting events (e.g. received characters) at once.

smx_SemDelete

BOOLEAN smx_SemDelete (SCB_PTR* shp)

Type SSR

Summary Deletes a semaphore created by smx_SemCreate().

Compl smx_SemCreate()

Parameters shp Semaphore handle pointer.

Returns TRUE Semaphore deleted.
FALSE Semaphore not deleted due to error.

Errors SMXE_INV_SCB Invalid semaphore handle.

Descr Resumes waiting tasks, giving them FALSE return values and deactivating their timeouts. Then clears the semaphore control block, releases it to the SCB pool, and sets *shp == smx_nullcb so it cannot be used again.

Example

```
SCB_PTR printer_avail;

smx_SemDelete(&printer_avail);
```

smx_SemPeek

u32 smx_SemPeek (SCB_PTR sem, SMX_PK_PAR par)

Type SSR

Summary Returns the current value of the parameter specified.

Parameters

sem	semaphore to peek.
par	What to return.

Returns

value	Value of par.
0	Value, unless error.

Errors

SMXE_INV_SCB	Invalid semaphore handle.
SMXE_INV_PAR	Argument not recognized.

Descr This service can be used to peek at a semaphore. Valid arguments are:

SMX_PK_FIRST	First task waiting on this sem.
SMX_PK_LAST	Last task waiting on this sem.
SMX_PK_MODE	Semaphore mode.
SMX_PK_COUNT	Current count.
SMX_PK_LIMIT	Limit.
SMX_PK_NAME	Name.

Example

```
SCB_PTR sem;
TCB_PTR top_task;

top_task = (TCB_PTR)smx_SemPeek(sem, SMX_PK_FIRST);
```

smx_SemSet

BOOLEAN smx_SemSet (SCB_PTR sem, SMX_ST_PAR par, u32 v1, u32 v2)

Type SSR

Summary Provides semaphore control.

Compl smx_SemPeek()

Parameters

sem	Semaphore to set.
par	Parameter to set.
vl	Value 1.
v2	Value 2.

Returns

TRUE	Parameter has been set.
FALSE	Parameter has not been set due to error.

Errors

SMXE_INV_PAR	par not recognized.
SMXE_INV_SCB	Invalid semaphore handle.
SMXE_PRIV_VIOL	Privilege violation; cannot call from umode (SecureSMX).

Descr par is of type SMX_ST_PAR. Available parameters are:

SMX_ST_CBFUN	Semaphore signal callback function = v1.

Loads the signal callback function into the semaphore control block. Using this service is highly recommended over directly setting internal semaphore modes, which may result in incorrect settings due to preemption of the current task. Also, direct semaphore mode setting is not possible in umode.

Example

```
u32 sema_ctr;

smx_SemSet(sema, SMX_ST_CBFUN, sema_cbfun);

void sema_cbfun(SCB_PTR sem)
{
    sema_ctr++;
}
```

This example loads sema_cbfun() into the sema control block. Each time sema is signaled, sema_ctr is incremented. See smx_SemSignal() for an example of semaphore callback function usage.

smx_SemSignal

BOOLEAN smx_SemSignal (SCB_PTR sem)

Type SSR

Summary Signals a semaphore.

Compl smx_SemTest(), smx_SemTestStop()

Parameters

sem	Semaphore to signal.	

Returns

TRUE	Signal sent.
FALSE	Error.

Errors

SMXE_INV_SCB	Invalid semaphore handle or mode.
SMXE_SIG_CTR_OVFL	Event or threshold counter has exceeded 255.

Descr

Mode	Action
RSRC:	Resume top task with TRUE if count >= lim, else count++.
EVENT:	Resume top task with TRUE if count >= lim, else: If lim == 1, count = 1. If lim != 1 and count < 255, count++.
THRES :	Resume top task and count -= lim if count >= lim, else: If count < 255, count++.
GATE :	Resume all waiting tasks with TRUE.

If sem->cbfun is not NULL, the callback function cbfun(SCB_PTR sem) is called. For a multiwait example see the smx_MsgXchgSet() example.

Example

```
TCB_PTR     calc;
XCB_PTR     in_xchg;
SCB_PTR     get_msg;
u32         get_msg_ctr = 0;

void sem_init(void)
{
    get_msg = smx_SemCreate(SMX_SEM_EVENT, 1, "get_msg");
     smx_SemSet(get_msg, SMX_ST_CBFUN, (u32)msg_count_cbf);
}
```

```
void calc_main(u32)
{
    u8* dp;

    while (1)
    {
        if(smx_MsgReceive(in_xchg, &dp, 10))
        {
            ProcessMsg(dp);
            smx_SemSignal(get_msg);
        }
        else
            /* report message failure */
    }
}

void msg_count_cbf(SCB_PTR sem)
{
    if (sem == get_msg)
        get_msg_ctr++;
}
```

In this example, sem_init() creates get_msg binary event semaphore and loads msg_count_cbf into its control block. calc_main() is called when the calc task starts running. Waits at in_xchg for a message. If a message is received in less than 10 ticks, calc processes the message, then signals get_msg. As a result, msg_count_cbf() is called, which increments get_msg_ctr. Although simplistic, this illustrates a method to keep track of received messages without creating a task to wait at the get_msg semaphore.

smx_SemTest

BOOLEAN smx_SemTest (SCB_PTR sem, u32 timeout=0)

Type SSR

Summary If sem has a pass condition, continues smx_ct. Otherwise, suspends it on sem.

Compl smx_SemSignal()

Parameters

sem	Semaphore to test.
timeout	Timeout in ticks or msec if |SMX_FL_MSEC.

Returns

TRUE	Test passed.
FALSE	Error or timeout.

Errors

SMXE_INV_SCB	Invalid semaphore handle.
SMXE_WAIT_NOT_ALLOWED	Called from an LSR.

Descr

mode:	action:
RSRC:	If sem->count > 0, decrement count and continue smx_ct with TRUE. Else, if timeout > 0, suspend smx_ct on sem. If timeout == 0, continue current task with FALSE.
EVENT:	Same.
GATE:	Same.
THRES :	Same, except if sem->count >= lim: count -= lim.

Waits forever if timeout == INF. Otherwise, if the timeout elapses before a pass condition occurs, waiting task resumes with FALSE. Operation from an LSR is the same as from a task, except that waits are not allowed.

Note 1. Clears smx_lockctr if called from a task and timeout != SMX_TMO_NOWAIT.

Example 1

```
SCB_PTR  start_cycle, data_ready;
TCB_PTR  get[N], process;
u8* name[] = ("get0", "get1", ..., “getN”);

start_cycle = smx_SemCreate(GATE, 1, "start_cycle");
data_ready = smx_SemCreate(THRES, N, "data_ready");

void init_main(u32)
{
    process = smx_TaskCreate(process_main, PR2, 200, 0, name[N]);
    smx_TaskStart(process);

    for (i = 0; i < N; i++)
    {
        get[N] = smx_TaskCreate(get_main, PR2, 200, 0, name[N]);
        smx_TaskStart(get[N]);
    }
    smx_SemSignal(start_cycle);
}

void get_main(u32)
{
    do
    {
        /* process data and store it globally */
        smx_SemSignal(data_ready);
    } while (smx_SemTest(start_cycle, TMO));

    /* notify of timeout or error */
}
```

```
void process_main(u32)
{
    while(1)
    {
        smx_SemTest(data_ready, INF))
        /* process global data */
        smx_SemSignal(start_cycle);
    }
}
```

In this example, there are N get tasks and one process task. After being created and started, the process task waits at the data_ready threshold semaphore, and each get task processes data and stores it, then signals the data_ready threshold semaphore and waits at the start_cycle gate semaphore. After N signals to the data_ready threshold semaphore, the process task is resumed. It processes the data, then signals the start_cycle gate semaphore, causing all of the get tasks resume operation.

If the get tasks need to wait on data or resources, multiple get tasks will result in more efficient usage of the processor than one get task attempting to get all of the data, since some get tasks can run while others are waiting.

Example 2

```
SCB_PTR  printer_ready;
TCB_PTR  t2a, t3a;

printer_ready = smx_SemCreate(RSRC, 1, "printer_ready");


void t2a_main(u32)
{
    ...
    smx_SemTest(printer_ready, TMO);
    /* send data to printer */
    smx_SemSignal(printer_ready);
}

void t3a_main(u32)
{
    ...
    smx_SemTest(printer_ready, TMO);
    /* send data to printer */
    smx_SemSignal(printer_ready);
}
```

This example shows sharing a printer between two tasks by using the printer_ready binary resource semaphore. Every task accessing the printer must first test this semaphore in order to avoid conflicts.

smx_SemTestStop

void smx_SemTestStop (SCB_PTR sem, u32 timeout=0)

Type Limited SSR — tasks only

Summary Operates the same as smx_SemTest(), except that ct is always stopped.

Compl smx_SemSignal()

Parameters

sem	Semaphore to test.
timeout	Timeout in ticks or msec if |SMX_FL_MSEC.

Errors

SMXE_INV_SCB	Invalid semaphore handle.
SMXE_OP_NOT_ALLOWED	Called from an LSR.

Descr See smx_SemTest() for operational description. smx_ct stops instead of suspending, then restarts instead of resuming. Pass or fail is returned via the parameter in taskMain(par), when task restarts.

Notes
1. If called from an LSR, aborts operation and returns to LSR.
2. Clears smx_lockctr if called from a task, since it always stops.

TaskMain void task_main(BOOLEAN par)

par

TRUE	Got semaphore.
FALSE	Error or timeout.

Example

```
SCB_PTR data_ready, start_cycle;
TCB_PTR  get[N], process;
u8* name[] = ("get0", "get1", ..., “getN”);

start_cycle = smx_SemCreate(GATE, 1, "start_cycle");
data_ready = smx_SemCreate(THRES, N, "data_ready");

void init_main(u32)
{
    process = smx_TaskCreate(process_main, PR2, 200, 0, name[N]);
    smx_TaskStart(process);

    for (i = 0; i < N; i++)
    {
        get[N] = smx_TaskCreate(get_main, PR2, 0, 0, name[N]);
        smx_TaskStart(get[N], 1);
    }
}
```

```
void get_main(BOOLEAN pass)
{
	if (pass)
	{
		/* process data and store it globally */
		smx_SemSignal(data_ready);
		smx_SemTestStop(start_cycle);
	}
	else
		/* notify of timeout or error */
}

void process_main(u32)
{
	while(1)
	{
		smx_SemTest(data_ready, INF))
		/* process global data */
		smx_SemSignal(start_cycle);
	}
}
```

This is equivalent to example 1 for smx_SemTest(), using one-shot get tasks. Note that the get tasks are created with no stacks and also that each runs as soon as it is started, because pass == 1. Each get task gets and stores data, signals the data_ready threshold semaphore, and then does a test stop at the start_cycle gate semaphore. While stopped, none of the get tasks requires a stack.

If the get tasks do not need to wait for inputs, one stack will suffice for all of them, since only one can run at a time. However, if a get task might need to wait for inputs, then achieving efficient operation requires more than one stack. If a get task cannot get a stack, it simply waits for a stack and the scheduler passes over it. Hence, the number of stacks needed can be optimized for the probability of waiting for inputs.

smx_SSR

See the smx User's Guide, Service Routines chapter for usage information and more examples.

smx_SSR_ENTER

void smx_SSR_ENTERn (u32 id, u32 p1, ... , u32 pn)

Type Macros if SMX_CFG_MACROS, else functions

Summary Used to begin a system service routine (SSR).

Compl smx_SSR_EXIT()

Parameters

id	SSR ID — see xdef.h.
p1-n	Parameters of the call.

Returns none

Descr All system service routines (SSRs) must begin with smx_SSR_ENTERn(), which first increments smx_srnest to block other SSRs. It then sets smx_ct->err = SMXE_OK, saves the next program address of smx_ct in smx_ct->susploc, and logs the SSR in EVB if SMX_CFG_EVB and the smx_evben flag for the SSR is set (see smx User's Guide, Event Logging chapter, selective logging section).

Custom SSRs can be created. See smx User's Guide, Service Routines chapter, custom SSRs section. It is recommended to start by copying one that is similar.

Example

```
BOOLEAN NewSystemService(TCB_PTR task)
{
    smx_SSR_ENTER1(MY_CALL_ID, task);
    /* do my_function */
    return(smx_SSR_EXIT(TRUE, MY_CALL_ID));
}
```

This example shows the use of smx_SSR_ENTER1() and smx_SSR_EXIT() for a typical system service with one parameter, and which returns a BOOLEAN. In between, you can put any C statements. Although it is typical for SSRs to return a BOOLEAN or handle, it is not necessary to return anything. The return type of an SSR may be void, in which case it will end with just smx_SSR_EXIT(0, id), with no return.

smx_SSR_EXIT

u32 smx_SSR_EXIT (ret, id)

Type Function

Summary Used to end a system service routine (SSR).

Compl smx_SSR_ENTERn() or smx_SSREntern().

Parameters

ret	Value to return.
id	SSR ID.

Returns Returns ret if the SSR call does not wait or if the wait times out (normally FALSE or 0). If wait does not time out, returns the value that is supplied by the complementary SSR (e.g. smx_MsgSend()).

Descr For a suspend call, if smx_srnest > 1, returns ret to the point of call. For a stop call, passes ret as the task main function parameter.

If smx_srnest == 1, tests if smx_lqctr > 0 or if smx_sched > 0. If so, branches to the prescheduler, which calls the LSR scheduler or the task scheduler or respectively. If not, decrements smx_srnest and does the same as smx_srnest > 1, above. An SSR must end with:

```
return(smx_SSR_EXIT(ret, id));
```

if there is a return value, or:

```
smx_SSR_EXIT(0, id);
```

if there is no return value (i.e. void). All intermediate returns and error exits also must call smx_SSR_EXIT().

Example See example above.

smx_Sys

See the smx User's Guide for discussion and more examples of smx_Sys services.
See the smxBase User's Guide for time measurement functions (e.g. sb_TM_START).

smx_SysEtimeGet

u32 smx_SysEtimeGet (u32 flags=0)

Type Bare macro

Summary Gets the current elapsed time in ticks or msec.

Parameters flags 0 for ticks, SMX_FL_MSEC for msec.

Returns smx_etime

Descr Returns etime in ticks or msec. Msec is for special uses such as demos and is not recommended for general use due to overflow for large values of etime.

Example

```
u32 etime;

etime = smx_SysEtimeGet();
```

smx_SysPseudoHandleCreate

void* smx_SysPseudoHandleCreate (void)

Type Bare function

Summary Creates a pseudo handle to identify an object that does not have a handle.

Parameters none

Returns pseudo handle
0 if no more pseudo handles available.

Descr Creates a pseudo handle to identify objects that do not have handles, such as ISRs and user-defined events. These can be used to log ISRs and user-defined events in EVB. Pseudo handles are also used by smxAware. They are in the range of SMX_PSEUDO_HANDLE_MIN to SMX_PSEUDO_HANDLE_MAX, in xdef.h. Each new pseudo handle is 4 greater than the previous one created.

Example See smx_EVB_LOG().

smx_SysPeek

u32 smx_SysPeek (SMX_PK_PAR par)

Type SSR

Summary Returns the current value of the parameter specified.

Parameters par What to return.

Returns value Value of argument.
0 Value, unless error.

Errors SMXE_INV_PAR argument is not recognized.

Notes This service can be used to peek at system variables. Valid arguments are:
SMX_PK_SEC Ticks per second

Example

```
u32 mspt;   /* milliseconds per tick */
mspt = 1000/smx_SysPeek(SMX_PK_SEC);
```

smx_SysStimeGet

u32 smx_SysStimeGet (void)

Type Bare macro

Summary Gets current system time in seconds.

Parameters none

Returns smx_stime

Descr Returns stime in seconds from its initial value.

Example

```
u32 stime;
stime = smx_SysStimeGet();
```

smx_SysPowerDown

BOOLEAN smx_SysPowerDown (u32 sleep_mode)

Type SSR

Summary Puts processor into sleep mode. Restores all tick-related timing when power resumes.

Parameters sleep_mode Sleep mode.

Returns TRUE Processor slept until awakened.
FALSE sleep_mode == 0.

Errors None

Descr If sleep_mode > 0, enters SSR and calls sb_PowerDown(sleep_mode). This is a user-implemented function, which saves the tick counter count and puts the processor into the desired sleep mode. Upon resumption of operation, sb_PowerDown() determines how many tick counter clocks have elapsed, calculates and loads the new tick counter value and returns the number of ticks lost.

smx_SysPowerDown() tests the first timer in smx_tq and the next task to timeout. It determines which of these events would have occurred first and if that event would have occurred during power down. If so, it performs the timeout operation for that event, then searches to find the next oldest event during power down and performs the timeout operation for it. This continues until all events, which would have occurred during power down, have occurred. The result is that LSRs and tasks are enqueued to run in the order they would have run, had power interruption not occurred.

The tick recovery process is not dependent upon the time lost, but rather upon how many timeouts would have occurred during that time. Hence, it can be effectively used in applications where long power interruptions occur. Cyclic and pulse timer events are requeued, when processed. If they reoccur within the power-down time, they will again be processed normally. Therefore, these timers will appear to operate normally, provided that smx_lq is large enough to handle all LSR invocations. If not, older LSR invocations will be lost.

After tick recovery is complete, stime is updated and the smx_SysPowerDown() is exited. Following this, LSRs then tasks will execute in the order invoked, resumed, or restarted. Since interrupts are enabled during smx_SysPowerDown(), smx_TickISR() can run and can invoke smx_KeepTimeLSR, which will run after other LSRs have run. Thus, new ticks will not be lost and LSRs will run in their order of occurrence.

Example

```
void smx_IdleMain(u32)
{
    while(TRUE)
    {
        ...
        if (idle_done)
            smx_SysPowerDown(SLEEP);
    }
}
```

This is the normal use of smx_SysPowerDown() — i.e. at the end of idle, after it has completed all of its work. At this point there is no useful work left to do, hence the processor can be put into SLEEP mode. Of course, once the processor is put into SLEEP mode, it is then dependent upon an event or interrupt to wake it up.

smx_SysWhatIs

SMX_CBTYPE smx_SysWhatIs (void* h)

Type SSR

Summary Returns control block type for handle.

Parameters h Handle.

Returns type Type of control block.
0 Control block type is not recognized.

Errors SMXE_INV_PAR Invalid handle.

Descr Returns the control block type of the control block pointed to by h. Returns NULL if h does not point to a valid control block. h is not range checked, so it is possible that it may return an invalid cbtype. It is advisable to check that the handle is in range for the cbtype returned before using the cbtype.

Example

```
SCB_PTR  sx;

sx = smx_SemCreate(SMX_SEM_RSRC, 1, "sem");
...
if (smx_SysWhatIs(sx) == SMX_CB_SEM)
    smx_SemSignal(sx);
```

smx_Task

See the smx User's Guide, Tasks chapter for usage information and more examples.

smx_TaskBump

BOOLEAN smx_TaskBump (TCB_PTR task, u8 pri)

Type SSR

Summary Changes task priority, unless pri == SMX_PRI_NOCHG, and requeues the task.

Parameters

task	Task whose priority to change.
pri	New priority, unless SMX_PRI_NOCHG.

Returns

TRUE	Task priority changed.
FALSE	Error.

Errors

SMXE_BROKEN_Q	Task queue is broken.
SMXE_INV_PRI	pri > SMX_MAX_PRI.
SMXE_INV_TCB	Invalid task handle.

Descr Changes task->normpri = pri, and if task owns no mutexes, task->pri = pri, otherwise task->pri is promoted, but not demoted. If pri == SMX_PRI_NOCHG, no priority changes are made. Whether or not task->pri is changed, if task is in smx_rq, it is requeued at the end of its priority level and it will preempt smx_ct if it is now the top task, or if task is in a priority queue, it will be requeued at the end of tasks with the same priority. If task is waiting for a mutex, the mutex owner's priority is promoted, if it is less than pri and priority inheritance is enabled for the mutex. If task is in a FIFO queue, it is not moved.

The current task can bump itself, which can result in it being preempted.

Example

```
void taskA_main(u32)
{
    smx_TaskUnlock();

    while (1)
    {
        /* do main function */
        smx_TaskBump(smx_ct, SMX_PRI_NOCHG)
    }
}
```

Each time taskA completes its main function, it bumps itself to the end of its priority level in smx_rq. This allows other tasks at the same priority level in smx_rq to run. If they also bump themselves to the end, round-robin or cooperative multitasking results.

smx_TaskCreate

TCB_PTR smx_TaskCreate (FUN_PTR fun, u8 pri, u32 tlssz_ssz, u32 fl_hn, const char* name=NULL, u8* sbp=NULL , TCB_PTR* thp=NULL)

Type SSR

Summary Creates a task with fun as its main function and with the parameters specified.

Compl smx_TaskDelete()

Parameters

fun	Main function: void fun(u32 par)
pri	Priority.
tlssz_ssz	Task Local Storage (TLS) size (high 16 bits) and stack size (low 16 bits)
fl_hn	Flags and the heap number, hn (low 4 bits), for stack. Flags:

SMX_FL_CHILD	create child ptask.
SMX_FL_LOCK	start locked.
SMX_FL_NONE	no flags specified.
SMX_FL_UMODE	task runs in unprivileged mode.

name	Name to give task or NULL for none.
sbp	Stack block pointer for preallocated stack, NULL for none.
thp	Task handle pointer (see hp note in Notes and Restrictions).

Returns

handle	Task created.
NULL	Task not created due to insufficient resources or error.

Errors

SMXE_INV_PAR	SecureSMX. CHILD & UMODE flags both set.
SMXE_INV_PRI	pri > SMX_MAX_PRI.
SMXE_INSUFF_HEAP	Insufficient heap for permanent stack.
SMXE_OUT_OF_TCBS	TCB pool is empty.

Descr Gets a task control block from the TCB pool. fun() becomes the main function (i.e. the entry point) for the task and pri becomes its priority. For a *preallocated stack* (sbp != NULL) the stack size is calculated by:

```
stksz -= (SMX_CFG_STACK_PAD_SIZE + SMX_RSA_SIZE + tlssz);
```

and task->flags.stk_preall = 1. A preallocated stack can come from any source or be a standalone block.

If sbp == 0, the task stack block is allocated from heap hn. The block size to allocate is calculated by:

```
sbsz  = SMX_CFG_STACK_PAD_SIZE + stksz + SMX_RSA_SIZE + tlssz;
```

If allocation fails, the TCB is released and smx_nullcb is returned. Both a preallocated stack and a heap stack are permanently bound to the task, and task->flags.stk_perm is set.

If stksz == 0, the task will be given a stack from the stack block pool when it begins running. That stack is not permanently bound to the task and will be released back to the stack block pool when the task stops.

All stack blocks can have a stack pad above the stack, the stack, the Register Save Area (RSA) below the stack, and Task Local Storage, TLS, below the RSA. The stack pad size is determined by SMX_CFG_STACK_PAD_SIZE in acfg.h, RSA size is 32 for ARM-M, and TLS size is determined by the tlssz_ssz parameter. The stack bottom (task->sbp) is aligned on an SB_STACK_ALIGN boundary (8 bytes for ARM-M). As a consequence, the actual stack size may be 4 bytes less than expected.

SMX_FL_LOCK, sets the task's start locked flag. This causes the task to always start in the locked state. This is useful to prevent task initialization from being interrupted. When initialization is done the task can be unlocked with smx_TaskUnlock(). Start locked is also useful for one-shot tasks. Other task flags are set as follows: stack high water mark valid ON (shwm = 0), stack check ON, permanent stack OFF if stack size is 0, else ON, all others OFF. The specified task name is stored in the TCB. This is useful when debugging to confirm that one is looking at the correct TCB.

If stack scanning is enabled by SMX_CFG_STACK_SCAN in xcfg.h, the stack pad and stack are filled with the SB_STK_FILL_VAL defined in bdef.h. Task stacks are periodically scanned by idle, and task->shwm records the stack high-water mark. This is useful during debugging to see how much of the stack is actually being used. When a task is suspended or stopped, task->flags.stk_chk == 1, and if shwm > stack size, SB_STK_FILL_OVFL is reported and the error manager, smx_EM() runs.

The last step is to return the address of the TCB as the task handle. This handle identifies the task and is used whenever the task is referred to. It should be stored in a global variable named for the task.

Notes:

1. Allocating the task stack from heap hn is a convenience during initialization. However, while running, if the heap is busy, the heap allocation may be forced to wait up to smx_htmo ticks on the heap mutex. If a timeout occurs, task create will fail. If the heap mutex is released before timeout, the task stack will be allocated, unless a large enough block cannot be found, and task create should succeed. Even if the heap mutex is free initially, the heap allocation may take some time, and task creation may be slower than expected. Using preallocated stacks during runtime will get around these problems. Another solution is to do task creation from low-priority tasks so critical tasks are not delayed.
2. Child pmode tasks can be created for non-MPU systems and the same limitations apply to them as listed below.

SecureSMX with SMX_CFG_MPU

The SMX_FL_CHILD flag can be used by a ptask to create a child ptask, and the SMX_FL_UMODE flag can be used by a ptask to create a utask. Both flags cannot be true in pmode – an SMXE_INV_PAR error results. Consequently, a ptask cannot create a umode child task. In umode, neither flag is necessary – any task created by a utask is automatically a child utask.

A task being created is given the default MPA, *mpa_dflt*, created by the user. The MPA can then be changed using mp_MPACreate(). (See the SecureSMX User's Guide.) Normally, during debug, mpa_dflt allows access to a wide range of memory. But the release version should permit no memory access, thus forcing a specific MPA to be assigned to the new task.

For v7M, an MPU region is automatically created for a preallocated stack block or a heap stack block and is loaded into MPA[7] when the task's MPA is created. (Region information is passed from smx_TaskCreate() to mp_MPACreate() using a few TCB fields that are not

yet needed.) The same is done by the scheduler when a stack is allocated from the stack block pool.

For v8M, if umode == 0 and a permanent task stack comes from heap0 in sys_data, no region is created for it when a task is created. Similarly, if a temporary stack comes from the stack pool in sys_data, no region is created for it when a task is dispatched. For more information see the Introduction and Getting Started sections of the SecureSMX User's Guide.

Example

```
TCB_PTR taskA, taskB;

void taskX_main(u32)
{
    smx_TaskLock();
    taskA = smx_TaskCreate(taskA_main, PRI_HI, 0, SMX_FL_NONE, "taskA");
    taskB = smx_TaskCreate(taskB_main, PRI_NORM, 1000, SMX_FL_NONE, "taskB");
    smx_TaskStart(taskB);
    smx_TaskStart(taskA);
    smx_TaskUnlock();
}
```

The above code creates two tasks and starts them. taskA has normal priority. It will be assigned a stack pool stack when it is dispatched. taskB has low priority. It is permanently bound to a 1000 byte stack from the heap. The task doing the initialization is locked so that tasks A and B will not preempt it until it is done. As a consequence, even though taskA is started after taskB, it will run first because it has higher priority. If taskX had low priority and were not locked, taskB would run first until it suspended or stopped, then taskX would run and start taskA. When taskA suspended or stopped, taskX would run and autostop.

smx_TaskCurrent

TCB_PTR smx_TaskCurrent (void)

Type Function

Summary Returns the current task handle.

Parameters none

Returns handle Current task.

Errors none

Descr It is preferrable to use this function in application code rather than reading smx_ct directly, which cannot be done from umode under SecureSMX.

smx_TaskDelete

BOOLEAN smx_TaskDelete (TCB_PTR* thp)

Type SSR

Summary Releases resources owned by the specified task, then deletes it.

Compl smx_TaskCreate()

Parameters

thp	Task handle pointer.	

Returns

TRUE	Task deleted.
FALSE	Task not deleted due to error.

Errors

SMXE_INV_PAR	smx_ct was passed instead of a task handle.
SMXE_INV_TCB	Invalid task handle.
SMXE_STK_OVFL	Stack overflow for self-deleting task.

Descr Dequeues the task from queue it is in, if any. Next if thp->cbfun != NULL, calls the task's callback function:

```
thp->cbfun(SMX_CBF_DELETE)
```

It is recommended that this case of the task callback function be written to release all resources owned by the task and to do any other shutdown operations needed. Writing this case in parallel with the SMX_CBF_INIT case is a good way to avoid forgetting to release a resource when a task is deleted.

smx_TaskDelete() calls an internal function, smx_TaskFreeAll(), which frees all timers, smx blocks, messages, and mutexes owned by the task that were not freed by the callback, and it deactivates its timeout. This is a general-purpose function that traverses each control block table to find and free all owned control blocks. This likely to be slow in large systems. Furthermore, non-owned objects such as bare blocks and child tasks cannot be found and thus released.

If the above operations pass and the task's stack was not preallocated, smx_TaskDelete() frees it back to its heap, if permanent, or to the stack pool, if temporary. If stack free is successful, releases the task's TCB back to the TCB pool and sets *thp = smx_nullcb, so it cannot be used again.

If the task is deleting itself, smx_TaskDeleteLSR is invoked, instead of the preceding code. This LSR runs before the scheduler and before any other task, so it acts as an extension of smx_TaskDelete() outside of the task being deleted. It tests for stack overflow, then frees the stack. If stack free is successful, releases the TCB back to the TCB pool, sets *thp = smx_nullcb, and sets smx_sched = SMX_CT_DELETE, which causes the task scheduler to skip current task processing and dispatch the next task. If not successful, sets smx_sched = SMX_CT_NOP, which results in the task scheduler not running and the task autostopping. Task self-delete should always be the last statement before the final }. Any code after it may execute with unpredictable results.

If any operation performed by smx_TaskDelete() fails, the task's TCB and handle will still be valid, thus delete can be retried. This is advisable, because the failure may have been due to a heap mutex timeout. In the case of self-delete, retry must be done by another task.

Notes: 1. Preemptively deleting one task by another task runs the risk of damaging a shared structure such as a heap. Normally this is done only in a partitioned environment under SecureSMX where potential damage is limited to the partition in which the deleted task is located.

2. It generally is best for a task to delete itself, or to be deleted by a lower-priority task. This ensures that the target task is not in the midst of a sensitive operation.

Example

```
TCB_PTR  t2a = smx_TaskCreate(t2a_main, TP2, 0, 0, "t2a");
 smx_TaskSet(t2a, SMX_ST_CBFUN, (u32)t2a_CBF);
 smx_TaskStart(t2a);

void 2a_main(u32)
{
      /* perform operation */
      smx_TaskDelete(&t2a);
}

void t2a_CBF(u32 mode)
{
      switch (mode)
      {
            case SMX_CBF_INIT:
                  /* get t2a objects and memory */
                  break;
            case SMX_CBF_DELETE:
                  /* free t2a objects and memory */
      }
}
```

The above example shows creating and starting a one-shot task, t2a, with a callback function. When smx_TaskStart(t2a) runs, the scheduler calls t2a_CBF(SMX_CBF_INIT) which gets all objects and memory that t2a needs to perform its operation. When t2a finishes, it self-deletes with smx_TaskDelete(&t2a). This calls t2a_CBF(SMX_CBF_DELETE), which releases all objects and memory that were obtained for t2a during initialization. Having these two cases adjacent in one function helps to avoid resource leaks due to not freeing an object or memory that was allocated to t2a.

smx_TaskLocate

void* smx_TaskLocate (TCB_PTR task)

Type SSR

Summary Locates the queue which a task is in.

Parameters

task	Task to locate.

Returns

handle	pointer to queue task is in.
NULL	task is not in a queue or error.

Errors SMXE_BROKEN_Q No queue control block found.
SMXE_INV_TCB Invalid task handle.

Descr Returns a pointer to the queue that task is in, if it is in a queue, or NULL if it is not in a queue. If in smx_rq, returns a pointer to the top level. Aborts and reports SMXE_BROKEN_Q if no queue control block is found.

Example

```
BOOLEAN resume_task(TCB_PTR task, MCB_PTR ack_msg)
{
    CB_PTR  q;

    q = (CB_PTR)smx_TaskLocate(task);

    switch (q->cbtype)
    {
        case SMX_CB_XCHG:
            smx_MsgSend(ack_msg, (XCB_PTR)q);
            pass = TRUE;
            break;
        case SMX_CB_SEM:
            smx_SemSignal((SCB_PTR)q);
            pass = TRUE;
        default:
            pass = FALSE;
    }
    return(pass);
}
```

This function allows resuming a task if it is waiting at an exchange for a message or waiting at a semaphore for a signal. Otherwise, task is left alone.

smx_TaskLock

BOOLEAN smx_TaskLock (void)

Type Bare function

Summary Increments the lock counter, which blocks the current task from being preempted.

Parameters None

Returns TRUE ct locked.
FALSE ct is locked, but lock counter was not incremented.

Errors SMXE_EXCESS_LOCKS smx_lockctr == SMX_CFG_LOCK_NEST_LIMIT.

Descr Increments smx_lockctr up to SMX_CFG_LOCK_NEST_LIMIT. The current task is locked as long as the lock counter is non-zero.

CAUTION: All smx services that stop or suspend ct will break its lock. Also smx services that may suspend smx_ct will break its lock, unless NO_WAIT is specified, or the service is called from an LSR.

Note In order to output the excess locks error message, smx_lockctr is temporarily reduced to 1, then put back to SMX_CFG_LOCK_NEST_LIMIT.

Example

```
u32 hour;

void hourly_main(u32)
{
    smx_TaskLock()
    hour++;
    smx_TaskUnlock()
}
```

In this example, other tasks are blocked from accessing hour while it is being updated.

smx_TaskLockClear

BOOLEAN smx_TaskLockClear (void)

Type SSR

Summary Clears the lock counter, thus allowing the current task to be preempted.

Parameters none

Returns TRUE Lock counter cleared and ct unlocked.
FALSE ct is unlocked, but lock counter was not 1.

Errors SMXE_INSUFF_UNLOCKS smx_lockctr is cleared, but was > 1.

Descr Clears smx_lockctr and tests for preemption. Recommended to be called instead of smx_TaskUnlock(), at the end of lock nesting in the task main function, as a precaution to ensure that the lock counter is zero.

Example

```
u32 hour;

void hourly_main(u32)
{
    smx_TaskLock()
    hour++;
    smx_TaskLockClear()
}
```

In this example, other tasks are blocked from accessing hour while it is being updated. Using this lock clear to unlock ensures that the task will be unlocked even if smx_lockctr > 1.

smx_TaskPeek

u32 smx_TaskPeek (TCB_PTR task, SMX_PK_PAR par)

Type SSR

Summary Returns the current value of the parameter specified.

Parameters

task	Task to peek at.
par	What to return.

Returns

value	Value of par.
0	Value, unless error.

Errors

SMXE_INV_TCB	Invalid task handle.
SMXE_INV_PAR	Parameter not recognized.

Descr This service allows peeking at a task. Valid arguments are:

SMX_PK_ERROR	Error last reported for task.
SMX_PK_FUN	Task main function.
SMX_PK_HN	Heap number for stack from heap, NULL if not.
SMX_PK_INDEX	Index of task in TCB pool. Also used for timeout array.
SMX_PK_LOCK	Lock counter count.
SMX_PK_MTX	First owned mutex.
SMX_PK_NAME	Name.
SMX_PK_NEXT	Next task linked to task in a queue; NULL if none.
SMX_PK_PARENT	Task's parent; NULL if none.*
SMX_PK_PREV	Previous task linked to task in a queue; NULL, if none.
SMX_PK_PRI	Priority.
SMX_PK_PRINORM	Normal priority.
SMX_PK_PRIV	Privilege level.*
SMX_PK_RTC	Runtime counter.
SMX_PK_RTLIM	Runtime limit.*
SMX_PK_RTLIMCTR	Runtime limit counter.*
SMX_PK_STATE	State.
SMX_PK_TLSP	Task local storage pointer.
SMX_PK_TMO	Timeout remaining.
SMX_PK_UMODE	Task is in umode.*

* means available only if SMX_CFG_MPU is TRUE under SecureSMX.

Example

```
TCB_PTR  atask;
u32  time_left;

time_left = smx_TaskPeek(atask, SMX_PK_TMO);
if (time_left > 10)
    smx_TaskResume(atask);
```

Resume atask, if it has more than 10 ticks left to wait.

smx_TaskResume

BOOLEAN smx_TaskResume (TCB_PTR task)

Type SSR

Summary Dequeues task from any queue it may be in and puts it into the ready queue at the end of its priority level.

Compl smx_TaskSuspend()

Parameters task Task to resume.

Returns TRUE Task resumed.
FALSE Task not resumed due to error.

Errors SMXE_INV_TCB Invalid task handle.

Descr Dequeues task from any queue it may be in and disables its timeout. If in an event queue the differential count of the following task is increased by the differential count of task. Task is resumed with 0, as if a timeout had occurred. Hence, the call on which it suspended will fail.

The current task may resume itself. The net result is that it is moved to the end of its smx_rq level. If the current task is still the top task in smx_rq or if it is locked, it is continued. Otherwise, it is preempted. Either way, it returns with TRUE.

smx_TaskResume() can be used for both suspended and stopped tasks. For example, if task had been suspended by smx_SemTest() it will be resumed, but if it had been stopped by smx_SemTestStop() it will be restarted.

Example

```
TCB_PTR  taskn;

void taskn_main(u32)
{
    do
    {
        /* perform taskn operations */
    } while (smx_TaskResume(smx_ct));
}
```

If other equal priority tasks are written this way and are in smx_rq, each will run, then move itself to the end of the rq level by calling smx_TaskResume(smx_ct). Higher priority tasks can preempt the round-robin tasks, but lower priority tasks are locked out. Note similarity to smx_TaskBump().

smx_TaskSet

BOOLEAN smx_TaskSet (TCB_PTR task , SMX_ST_PAR par, u32 v1=0, u32 v2=0)

Type SSR

Summary Provides task control.

Compl smx_TaskPeek().

Parameters

task	Task to modify.
par	Parameter to set.
v1	First value.
v2	Second value.

Returns

TRUE	Parameter has been set.
FALSE	Parameter not set due to error.

Errors

SMXE_INV_PAR	par not recognized.
SMXE_INV_TCB	Invalid task handle.
SMXE_PRIV_VIOL	Privilege violation; cannot call from umode (SecureSMX).

Descr Used to modify task operation. par is of type SMX_ST_PAR. Available parameters are:

SMX_ST_CBFUN	Set task callback function = v1 & set task->flags.hookd, if v2 >0.
SMX_ST_FUN	Set task main function = v1.
SMX_ST_IRQ	Set task IRQ permission struct pointer = v1.*
SMX_ST_PRITMO	Set task timeout priority = v1, save previous priority @v2.
SMX_ST_PRIV	Set task privilege level = v1.*
SMX_ST_RTLIM	Set top parent task runtime limit = v1 and runtime counter = 0.**
SMX_ST_STK_CK	Set task stack check flag = v1.
SMX_ST_STRT_LOCKD	Set task start locked flag = v1.
SMX_ST_TAP	Set top parent task token array pointer = v1.*
SMX_ST_UMODE	Set task umode flag = v1.*

smx_TaskSet() can be called only from pmode. * means available only if SMX_CFG_MPU is set. ** means available only if SMX_CFG_RTLIM is set.

SMX_ST_FUN can be used to change a task's main function. However smx_TaskStartNew() is preferred for this since it allows also changing the parameter for the main function and the task priority. In addition, it restarts the task.

For SMX_ST_PRITMO, task's current normal priority is stored in the location pointed to by v2. This is used to restore normal priority after timeout processing is complete. To set task priority use smx_TaskBump().

Units for SMX_ST_RTLIM are clocks of the timer that generates the smx tick. task->rtlim and task->rtlimctr can be set only in the top parent task. Child tasks have pointers in these fields to the corresponding fields in their top parent tasks.

This service is an SSR. Using it is highly recommended vs. directly setting TCB flags and fields, which may result in incorrect settings due to preemption of the current task. If par is not recognized, returns FALSE.

Example 1:

```
smx_TaskSet(task, SMX_ST_STK_CK, ON);
```

This example turns on stack checking for overflow when task is suspended, stopped, or deleted. Stack checking must be disabled for any function which changes stacks, because if a preempt occurs during the function the smx stack check code will report false overflow errors. For example:

```
smx_TaskSet(smx_ct, SMX_ST_STK_CK, OFF);
/* call function which changes stacks */
smx_TaskSet(smx_ct, SMX_ST_STK_CK, ON);
```

Example 2:

```
smx_TaskSet(task, SMX_ST_CBFUN, task_cbf, 1);
```

This loads cbfun into task->cbfun and sets task->flags.hookd = 1. For information on how to use task callback functions see the smx User's Guide, Tasks chapter, task callback functions section.

smx_TaskSleep

BOOLEAN smx_TaskSleep (u32 time)

Type Limited SSR — tasks only

Summary Suspends the current task until the specified system time, stime.

Parameters

time	Time to awaken, in seconds from now.	

Returns

TRUE	ct has been delayed.
FALSE	No delay due to error.

Errors

SMXE_INV_TIME	time <= smx_stime
SMXE_OP_NOT_ALLOWED	Called from an LSR

Descr If time is greater than stime, smx_ct is suspended, and its timeout is set to

```
timeout = smx_etime + (time - smx_stime) * SMX_CFG_TICKS_PER_SEC
```

The amount added to etime must be less than 2^31. This allows sleeping up to 248 days for a 100 tick per second clock rate. When the task times out, it is resumed with TRUE. Resolution is one second.

Note Clears smx_lockctr if called from a task.

Example 1

```
smx_TaskSleep(smx_SysStimeGet() + 10);      /* sleep until 10 seconds from now */
```

Example 2

```
TCB_PTR hourly;
u32 stime = smx_SysStimeGet();
u32 next_hour = stime + (3600 - (stime % 3600));
smx_TaskStart(hourly);

void hourly_main(u32)
{
    while(smx_TaskSleep(next_hour))
    {
        /* perform hourly function */
        next_hour += 3600;
    }
}
```

In this example, the hourly task wakes up at the start of the next hour and performs its hourly function. It then performs its hourly function, every hour on the hour.

smx_TaskSleepStop

void smx_TaskSleepStop (u32 time)

Type Limited SSR — tasks only

Summary Stops the current task until the specified system time.

Parameters time Time to awaken in seconds from now.

Errors

SMXE_INV_TIME	time <= smx_stime
SMXE_OP_NOT_ALLOWED	Called from an LSR

Descr See smx_TaskSleep() for operational description. smx_ct always stops instead of suspending, then restarts instead of resuming. Pass or fail is returned via the parameter in taskMain(par), when task restarts.

TaskMain void task_main(BOOLEAN par)

par

TRUE	Current task has been delayed.
FALSE	No delay due to error.

Notes

1. If called from an LSR, aborts operation and returns to LSR.
2. Clears smx_lockctr if called from a task, since it always stops.

Example

```
TCB_PTR hourly;
u32 stime = smx_SysStimeGet();
u32 next_hour = stime + (3600 - (stime % 3600));
smx_TaskStart(hourly, FALSE);

void hourly_main(BOOLEAN pass)
{
    if (pass)
    {
        /* perform hourly function */
        next_hour += 3600;
    }
    smx_TaskSleepStop(next_hour);
}
```

This is the equivalent one-shot task for the previous example. In this example, next_hour is set equal to stime and hourly task is started with pass == FALSE. This prevents performing the hourly function, the first time. The hour task sets next_hour to the start of the next hour and sleeps until then. Pass == 1, from smx_TaskSleepStop() causes the hourly function to be performed and this process will repeat until stopped.

smx_TaskStart

BOOLEAN smx_TaskStart (TCB_PTR task, u32 par)
BOOLEAN smx_TaskStartNew (TCB_PTR task, u32 par, u8 pri, FUN_PTR fun)

Type SSR

Summary smx_TaskStart() starts or restarts task with par after it has been created.
smx_TaskStartNew() restarts task with par, pri, and fun.

Compl smx_TaskStop()

Parameters

task	Task to start.
par	Parameter to pass to task.
pri	New priority.
fun	New main function.

Returns

TRUE	Task started.
FALSE	Task not started due to error.

Errors

SMXE_INV_PRI	pri > SMX_PRI_NUM and !SMX_PRI_NOCHG.
SMXE_INV_TCB	Invalid task handle.

Descr Both Start()'s can be called from any task or LSR. Each dequeues task from any queue it may be in. If in an event queue, its differential count is added to that of the next task. If task is not smx_ct, task->sp is cleared, stack check is inhibited, and if not bound, its stack is freed (since it will get a new stack when dispatched by the scheduler). Then task is put into the ready queue, and its timeout is disabled.

If smx_ct starts itself, the result is that it is stopped and moved to the end of its smx_rq level. Its stack is later released by the scheduler. These actions occur even if smx_ct is locked. If it is still the top task in smx_rq, or locked, it is immediately restarted. Otherwise, it is preempted. In either case, code statements following any of the task starts do not execute.

If an LSR starts the current task, operation is as the same, except that the task start returns to the LSR, as it would to a task, other than smx_ct.

When task is restarted, it is given a new stack and it starts from the beginning of its main function with par as its parameter. Since task is restarting, it is not necessary to indicate that wait failed. It will be started locked, if its strt_lockd flag is set.

smx_TaskStart() is used primarily to start a new task or to restart a stopped task. Since it will restart any existing task, it may also be used to abort a task and restart it, even if the task is locked.

smx_TaskStartNew() loads fun into task->fun, loads pri into task->pri, normpri, and pritmo, unless pri == SMX_PRI_NOCHG.

If task->cbfun is not NULL, the callback function cbfun(SMX_CBF_INIT) is called. This function can be used to obtain all objects and memory that task needs to run. See also smx_TaskSet() and smx_TaskDelete().

Note See CAUTION in smx_TaskStop().

Example 1

```
LCB_PTR  tx_LSR;
TCB_PTR  tx_task;

void tx_ISR(void)
{
      if (xmit_complete)
      {
            smx_LSR_INVOKE(tx_LSR, 0);
      }
}

void tx_LSR_main(void)
{
      smx_TaskStart(tx, 0);
}
```

```
void tx_task_main(u32 timeout)
{
    if(timeout)
        /* resend message */
    else
        /* send next message */
    smx_TaskStop(smx_ct, TX_TIMEOUT);
}
```

tx_LSR is invoked by tx_ISR when a message transmission is complete. It restarts the tx task with timeout == 0, causing it to send the next message. If the message is not transmitted in time, the delay will complete and tx will restart with timeout = TRUE, causing it to resend the message.

Example 2

```
TCB_PTR  gp_task;

void appl_init(void)
{
    gp_task = smx_TaskCreate(gp_task_init, PRI_MAX, NO_STACK, SMX_FL_NONE,
                                                                    "gp_task");
    smx_TaskStart(gp_task);
}

void gp_task_init(u32)
{
    /* perform initialization */
    smx_TaskStartNew(smx_ct, 0, PRI_NORM, gp_task_run);
}

void gp_task_run(u32)
{
    /* perform normal operations */
}
```

In this example, the gp_task is initially started at maximum priority with gp_task_init() as its code. When initialization of gp_task is complete, smx_TaskStartNew() causes gp_task to start gp_task_run() with normal priority. This approach is commonly used for one-shot tasks, which require initialization.

smx_TaskStop

BOOLEAN smx_TaskStop (TCB_PTR task, u32 timeout=SMX_TMO_INF)

Type SSR

Summary Dequeues task, releases its stack if not a permanently bound stack, and sets its timeout to restart it after timeout ticks.

Compl smx_TaskStart()

Parameters

task	Task to stop.
timeout	Timeout in ticks or msec if |SMX_FL_MSEC.

Returns

TRUE	OK if task != smx_ct.
FALSE	Task not stopped due to error.

Errors SMXE_INV_TCB Invalid task handle.

Descr Dequeues task from any queue it may be in. If task is in an event queue, its differential count is added to that of the next task, if any. task->sp is cleared, and a stack pool stack is released to the freestack pool or, if SMX_CFG_STACK SCAN is TRUE, the stack is released to the scanstack pool and later moved to the freestack pool after it has been scanned and refilled with the test pattern.

If timeout > 0 task's timeout is set to timeout. If timeout == SMX_TMO_NOWAIT (0) or when the timeout elapses, task is put into smx_rq at the end of its priority level. If timeout == SMX_TMO_NOCHG, task's timeout is not changed.

This is the only system service which can stop another task and set its timeout. Hence, it can be used to cause another task to restart immediately or to restart after a timeout.

A task may also stop itself, even if it is locked. In this case, smx_lockctr() is cleared. If task stops itself, smx_TaskStop() is the last statement it executes. smx_ct may also be stopped by an LSR, even if is locked.

TaskMain void task_main(u32 par)

par

TRUE	task was stopped (cannot be FALSE).

Notes

1. During a stack scan, if the system stack high water mark exceeds the system stack size, SMXE_STK_OVFL is reported. This will normally occur during idle.
2. CAUTION: Preempting a task and stopping it is likely to cause damage. Even stopping a task that is not running could cause damage. The safest approach is for a task to stop itself. Using task callback functions, might make stopping a task by other tasks safe. See smx User's Guide, Tasks chapter, task callback functions section.

Example

```
TCB_PTR task;

void task_stop(TCB_PTR task)
{
    /* release all blocks, msgs, mutexes, and heap blocks owned by task */
    smx_TaskStop(task, SMX_TMO_INF);
}
```

task_stop() releases all objects that task owns, then stops it indefinitely. task ends up in a dormant state from which it can be restarted only by another task. A task callback function with case SMX_CBF_STOP could be defined to do the releases, then smx_TaskStop() could be called directly.

Task or LSR Autostop

```
u32  task_main(u32  par)
{
    ...
    return(par);
}
```

or

```
void  task_main(u32)
{
    ...
}
```

Parameters par Value passed to task if it is restarted.

Errors none

Descr When used in the main function of a task, return() or the final } have the same effect as smx_TaskStop(smx_ct, SMX_TMO_INF). If a return value is specified in return(), it is loaded into smx_ct->rv. Thus, a task can pass a value, such as a message handle, back to itself. Otherwise, smx_ct->rv is loaded with whatever value is in the register the C compiler uses to return a value.

When used in an LSR, return() or the final } return control to the LSR scheduler. Any return value is ignored.

Example 1

```
TCB_PTR  comm;

u32 comm_main(u32 bp)
{
    u8* dp = (u8*)bp;
    /* use dp as working pointer to access the block */
    return(bp);
}
```

In the above, comm accepts a block pointer passed to it by another task and passes this pointer back to itself each time it stops. In this way, an unbound task can preserve local information from one run to the next.

Example 2

```
void  task_main(u32 msg)
{
      …
      return((u32)smx_MsgReceive(input, 0, TMO));
}

void  task_main(u32 msg)
{
      …
      smx_MsgReceive(input, 0, TMO);
}

void  task_main(u32 msg)
{
      …
      smx_MsgReceiveStop(input, 0, TMO);
}
```

produce the same result — the current task is stopped, and the value returned by smx_MsgReceive() is passed to it. The last example waits for a message without a stack. The first two wait for a message with a stack. Since the stack is lost in all three cases, the last is the best way to implement the smx_MsgReceive().

smx_TaskSuspend

BOOLEAN smx_TaskSuspend (TCB_PTR task, u32 timeout=0)

Type SSR

Summary Dequeues task and sets its timeout to resume after timeout ticks.

Compl smx_TaskResume()

Parameters

task	Task to suspend. SMX_CT == smx_ct.
timeout	Timeout in ticks or msec if |SMX_FL_MSEC.

Returns

TRUE	Task suspended.
FALSE	Error.

Errors SMXE_INV_TCB Invalid task handle.

Descr Dequeues task from whatever queue it may be in. If task is in an event queue, its differential count is added to that of the next task, if any. If task is already suspended or stopped, this call has no effect, except to possibly change its timeout.

If timeout > 0 task's timeout is set to timeout. If timeout == SMX_TMO_NOWAIT (0) or when the timeout elapses, task is put into smx_rq at the end of its priority level. If timeout == SMX_TMO_NOCHG, task's timeout is not changed.

This is the only system service which can suspend another task and set its timeout. Hence it can be used to delay another task without restarting it. When the timeout elapses, the other

task will resume if it was suspended or restart if it was stopped. However, if the task was in a wait queue, it will be dequeued, and the call that put it there will fail.

If smx_ct is suspending itself or if it is suspended by an LSR, its run context is saved in its Register Save Area (RSA). When a task suspends itself, smx_TaskSuspend() is the last statement executed until the task is resumed after timeout. If smx_ct is locked, smx_lockctr is cleared. Hence, smx_ct no longer will be locked when it resumes.

Notes

1. smx_TaskSuspend(smx_ct, SMX_TMO_NOWAIT) is the only case of a NO_WAIT self-suspend that clears smx_lockctr. The reason for this is that it bumps smx_ct to the end of its ready queue level and thus smx_ct may actually be suspended.
2. CAUTION: Although preemptively suspending a task by another task may not directly damage a heap or other global structure, it should not be done for too long because it may result in resources being tied up by the suspended task.

Example

```
TCB_PTR taskA;

void function(void)
{
    smx_TaskSuspend(taskA, SMX_TMO_INF);
    smx_TaskSuspend(smx_ct, SEC);
    /* statements after this will not execute for one second */
    ...
}
```

In this example, the function suspends taskA, indefinitely, then suspends itself for a second. In so doing, it preserves the context and local variables of both tasks.

smx_TaskUnlock

BOOLEAN smx_TaskUnlock (void)

Type Bare function that calls SSR

Summary Decrements smx_lockctr. If it becomes 0, unlocks the current task and tests for preemption.

Parameters none

Returns

TRUE	Operation performed.
FALSE	smx_ct was already unlocked.

Errors SMXE_EXCESS_UNLOCKS

Descr Decrements smx_lockctr; if smx_lockctr is already 0, aborts and issues SMXE_EXCESS_UNLOCKS error; if it is already 1, calls smx_TaskLockClear() to clear smx_lockctr and to check if a higher-priority task is ready to run. If so, smx_ct is preempted.

Note Any smx function that might suspend or stop the current task will also clear smx_lockctr, whether or not suspension or stopping actually occurs. Thus protection is lost.

Example 1

```
u32 hour;

void hour_incr(void)
{
    smx_TaskLock()
    hour++;
    smx_TaskUnlock()
}
```

In this example, other tasks are blocked from accessing hour while it is being updated.

Example 2

```
void hourly_main(u32)
{
    smx_TaskLock()
    hour_incr();
    if (hour > 24)
        hour = 0;
    smx_TaskUnlock()
}
```

This example works with the previous example to show why lock nesting is necessary. The hour_incr() routine could be called alone, so it must be locked. But hourly_main() also needs to be locked. Using a counter handles this situation.

Example 3

```
smx_TaskLock();
smx_SemSignal(semA);
smx_MsgReceive(xchgA, &dp, tmo);
```

In this example, the task lock prevents ct from being preempted if there is a higher priority task waiting at semA. smx_MsgReceive() clears the lock, whether it waits or not. Use of the lock, in this way, prevents an unnecessary potential task switch.

smx_TaskUnlockQuick

BOOLEAN smx_TaskUnlockQuick (void)

Type Bare function

Summary Decrements smx_lockctr. If it becomes 0, unlocks smx_ct, but does not test for preemption.

Parameters none

Returns

TRUE	Operation performed.
FALSE	ct was already unlocked

Errors SMXE_EXCESS_UNLOCKS

Descr Decrements smx_lockctr. If smx_lockctr is already 0, aborts and issues SMXE_EXCESS_UNLOCKS warning. This function is intended for quick, protected

accesses to global variables where the overhead of an SSR is not desirable. If a higher priority task is ready, it will not run until the next SSR or LSR finishes.

Example

```
u32 hour;

void hourly_main(u32)
{
    smx_TaskLock()
    hour++;
    smx_TaskUnlockQuick()
}
```

In this example, other tasks are blocked from accessing hour while it is being updated. Using this version of unlock eliminates the overhead of an SSR, but a higher priority task may be kept waiting.

smx_TaskYield

BOOLEAN smx_TaskYield (void)

Type SSR wrapper

Summary Requeues the current task at the end of its level of the ready queue, to allow others at that level to run.

Parameters none

Returns TRUE Task yielded.
FALSE Task did not yield due to an error.

Errors SMXE_BROKEN_Q

Descr Calls smx_BumpTask(smx_ct, SMX_PRI_NOCHG). It is normally a macro, but for SecureSMX it is a function so it can access smx_ct when called from umode via an SVC call.

Example

```
TCB_PTR  taskA;

void taskA_main(u32)
{
    do
    {
        /* perform taskA function */
    } while(smx_TaskYield());

    /* fix broken queue */
}
```

taskA performs its function, then yields to other tasks at its priority level in smx_rq. When these tasks have performed their functions and yielded, taskA will run again, unless it failed to yield initially.

smx_Timer

See the smx User's Guide, Timers chapter for usage information and more examples.

smx_TimerDup

BOOLEAN smx_TimerDup (TMRCB_PTR* tmrbp, TMRCB_PTR tmra, const char* name=NULL)

Type SSR

Summary Creates a duplicate timer tmrb from tmra and enqueues it after tmra in tq.

Parameters

	tmrbp	Pointer to location for tmrb handle.
	tmra	Timer to duplicate.
	name	Name to give timerb or NULL for none.

Returns

TRUE	Timer duplicated.
FALSE	Timer not duplicated due to error

Errors

SMXE_INV_OP	Attempted multiple duplication of tmrb.
SMXE_INV_PAR	tmrbp == NULL.
SMXE_INV_TMRCB	Invalid tmrb handle.
SMXE_OUT_OF_TMRCBS	Out of timer control blocks.

Descr Gets a TMRCB for tmrb and copies all fields from tmra into it, except for name, diffcnt (differential count), hptr (handle pointer), and onr. Then enqueues tmrb after tmra with tmrb->diffcnt == 0. Loads tmrbp into tmrb->hptr and tmrb into tmrbp. Loads the current LSR pointer or task handle into tmrb->onr. Hence, any task or LSR can duplicate a timer and will be identified as the owner of the duplicate timer. tmrb is effectively an exact duplicate of tmra and has all of the same properties, except as noted.

Example

```
LCB_PTR  lsra;
TMRCB_PTR tmra, tmrb;

smx_TimerStart(&tmra, 10, 0, lsra, "tmra");
smx_TimerDup(&tmrm, tmra, "tmrb");
```

In this example, tmrb is created as a duplicate of tmrs and it is enqueued in tq immediately after tmra with 0 differential count. tmrb can then be changed, if desired, by any of the services described below.

smx_TimerPeek

u32 smx_TimerPeek (TMRCB_PTR tmr, SMX_PK_PAR par)

Type SSR

Summary Returns the current value of the parameter specified.

Parameters

tmr	Timer to peek at.
par	What to return.

Returns

value	Value of par.
0	Value, unless error.

Errors

SMXE_INV_PAR	par not recognized.
SMXE_INV_TMRCB	Invalid timer handle.

Descr This service allows peeking at an active timer. Valid arguments are:

SMX_PK_COUNT	Number of timeouts since cyclic or pulse timer started.
SMX_PK_DELAY	Delay for next pulse HI or LO, if PULSE == LO or HI, resp.
SMX_PK_DIFF_CNT	Differential count from timer before.
SMX_PK_LSR	LSR to be invoked on timeout.
SMX_PK_LPAR	Parameter value to pass to LSR, if tmr->opt = SMX_TMR_PAR.
SMX_PK_MAX_DELAY	Total remaining time until timeout of last timer in tq.
SMX_PK_NAME	Name of timer.
SMX_PK_NEXT	Next timer in tq. NULL, if none.
SMX_PK_NUM	Number of timers in tq.
SMX_PK_ONR	Task or LSR that created tmr.
SMX_PK_OPT	LSR parameter option. (See smx_TimerSetLSR().)
SMX_PK_PERIOD	Period of cyclic or pulse timer.
SMX_PK_PULSE	Pulse state: LO or HI.
SMX_PK_TIME_LEFT	Total remaining time until timeout for tmr.
SMX_PK_WIDTH	Pulse width of pulse timer.

Example1

```
LCB_PTR     lsra;
TCB_PTR     taska;
TMRCB_PTR   tmra;

smx_TimerStart(&tmra, 5, 10, lsra, "tmra");
taska = (TCB_PTR)smx_TimerPeek(tmra, SMX_PK_ONR);
```

In this example, tmra is created. At some later time its owner task is determined.

Example2

```
void*       onr;
LCB_PTR     lsrb;

onr = (void*)smx_TimerPeek(tmra, SMX_PK_ONR);
if (smx_SysWhatIs(onr) == SMX_CB_TASK)
    taska = (TCB_PTR)onr;
else
    lsrb = (LCB_PTR)onr;
```

This example is for the case where tmra owner might be an LSR.

smx_TimerReset

BOOLEAN smx_TimerReset (TMRCB_PTR tmr, u32* tlp=NULL)

Type SSR

Summary Stops a timer then restarts it with its *current delay*. Saves its time left in tlp, unless NULL.

Parameters

tmr	Timer to reset.
tlp	Pointer to location to store time left.

Returns

TRUE	Timer restarted.
FALSE	Timer not restarted due to error.

Errors

SMXE_INV_TMRCB	Invalid timer handle.

Descr Dequeues tmr from the timer queue, tq. Its differential count is added to that of the next timer, if any. The total time remaining for tmr is computed and loaded into the location pointed to by tlp, unless tlp is NULL. Then requeues tmr in tq using its current delay and returns TRUE.

If the tmr is a one-shot timer, its current delay is its initial delay (i.e. the delay it was started with). For cyclic and pulse timers, the current delay is the initial delay until the first period starts. After that, for a cyclic timer, the current delay is the period, and for a pulse timer, the current delay is the delay until the end of the current HI or LO period.

If tmr has already timed out (i.e. tmr == NULL), returns FALSE and loads 0 into tlp, unless it is NULL. tmr cannot be restarted in this case because its TMRCB has already been cleared and returned to the TMRCB pool.

Example

```
TMRCB_PTR tmra;

smx_TimerStart(&tmra, 10, 0, lsra, "tmra");

while (1)
{
    while (wait_for_event()) {}
    /* perform actions */
    smx_TimerReset(tmra, NULL);
}
```

```
void lsra_mainu32)
{
    /* deal with timeout */
}
```

In this example, tmra is a 10 tick one-shot timer. Then the while loop waits for an event. When the event occurs, it performs the required actions, then resets tmra. If the next event does not occur within 10 ticks, tmra times out and invokes lsra to deal with the timeout. In this case, wait_for_event() is not an smx service, so it has no timeout.

smx_TimerSetLSR

BOOLEAN smx_TimerSetLSR (TMRCB_PTR tmr, LCB_PTR lsr, SMX_TMR_OPT opt, u32 par=0)

Type SSR

Summary Changes LSR, LSR option, and LSR parameter for the specified timer.

Parameters

tmr	Timer to change.
lsr	LSR.
opt	LSR option.
par	LSR par.

Returns

TRUE	Timer changed.
FALSE	Error. Timer not changed.

Errors

SMXE_INV_PAR	lsr == NULL or opt > 3.
SMXE_INV_TMRCB	Invalid timer handle.

Descr Loads new values for LSR, LSR option, and LSR parameter into the timer's TMRCB. The LSR option controls what is passed to the LSR when it is invoked:

SMX_TMR_PAR	par stored in TMRCB.
SMX_TMR_STATE	Pulse state: LO == 0, HI == 1.
SMX_TMR_TIME	etime at timeout.
SMX_TMR_COUNT	Number of timeouts since start.

These options help to reduce the need for peeks by the LSR. When a timer is started, the LSR option defaults to SMX_TMR_PAR and the LSR parameter defaults to 0. This service is used to change them, as well as the LSR, if desired. Note: The timeout counter is a 16-bit value, so it will rollover at 2^16 timeouts, if the cyclic or pulse timer runs that long.

Example

```
LCB_PTR  lsra;
TMRCB_PTR tmra;

smx_TimerStart(&tmra, 10, 10, lsra, "tmra");
smx_TimerSetLSR(tmra, lsra, SMX_TMR_COUNT, 0);
```

```
void lsra_main(u32 count)
{
    if (count < 100)
        /* perform function */
    else
        smx_TimerStop(tmra, NULL);
}
```

In this example, tmra is started, then it is modified to pass the timeout count to lsra, instead of tmra->par. After 100 timeouts, lsra stops tmra.

smx_TimerSetPulse

BOOLEAN smx_TimerSetPulse (TMRCB_PTR tmr, u32 period, u32 width)

Type SSR

Summary Changes period and pulse width for specified timer.

Parameters

tmr	Timer to change.
period	Timer period.
width	Pulse width.

Returns

TRUE	Timer changed.
FALSE	Timer not changed due to error.

Errors

SMXE_INV_PAR	Width >= period.
SMXE_INV_TMRCB	Invalid timer handle.

Descr Loads new values for timer period and pulse width into its TMRCB. These values do not take effect until the next period. For example, if this service is called in the middle of a pulse (state == HI), the pulse is allowed to complete normally and the inter-pulse period is allowed to complete normally, then the new width takes effect. Or if called in the middle of an inter-pulse period (state == LO), that state is allowed to complete normally, then the new width takes effect. The new period takes effect following the new width. This ensures smooth transitions for modulation techniques.

When a timer is started, its width is 0, by default. Hence this service converts a cyclic timer into a pulse timer if width > 0. Otherwise, it can be used to change the period of a cyclic timer, without having to restart the timer. Because the period or width or both can be changed, this service can be used for pulse width modulation (PWM), pulse period modulation (PPM), or frequency modulation (FM). See smx User's Guide, Timer chapter sections for more discussion of these.

Example

```
LCB_PTR  lsra;
TMRCB_PTR tmra;

smx_TimerStart(&tmra, 5, 10, lsra, "tmra");
smx_TimerSetPulse(tmra, 10, 5);
smx_TimerSetLSR(tmra, lsra, SMX_TMR_STATE, 0);
```

```
void lsra_main(u32 pulse)
{
    if (pulse == HI)
        Lamp(ON);
    else
        Lamp(OFF);
}
```

In this example, tmra is started, then changed to a pulse timer with a pulse width of 5 ticks and a period of 10 ticks (i.e. 5 ticks HI and 5 ticks LO). The timer is set to pass the pulse state to lsra when it changes state. This is used to turn a lamp on or off.

smx_TimerStart

BOOLEAN smx_TimerStart (TMRCB_PTR* tmrhp, u32 delay, u32 period, LSR_PTR lsr, const char* name=NULL)

BOOLEAN smx_TimerStartAbs (TMRCB_PTR* tmrhp, u32 time, u32 period, LSR_PTR lsr, const char* name=NULL)

Type SSR

Summary Creates and starts a new timer or restarts an existing timer.

Compl smx_TimerStop()

Parameters

tmrhp	Timer handle pointer.
delay	Timeout, in ticks, from now.
period	Period, if cyclic timer, 0 if not.
time	Absolute time from startup (i.e. etime == 0).
lsr	LSR to invoke at timeout.
name	Name to give timer or NULL for no name.

Returns

TRUE	Timer created and started or restarted.
FALSE	Timer not created due to error.

Errors

SMXE_INV_PAR	tmrhp == NULL, delay == 0, time < smx_etime, or lsr == NULL.
SMXE_INV_TMRCB	Invalid timer handle.
SMXE_OUT_OF_TMRCBS	Out of timer control blocks.

Descr If *tmrhp == NULL, a new timer is being started. A timer control block (TMRCB) is allocated from the TMRCB pool (smx_tmrcbs), and the start parameters: delay, period, lsr, and name are loaded into it. In addition, the TMRCB onr field is set to the current task or to the current LSR, depending upon which made this call. Other TMRCB fields are set to default values, which can be changed by other timer services.

If *tmrhp != NULL an existing timer is being restarted. The timer is dequeued from the timer queue, tq. Then the delay, period, lsr, and name fields in the TMRCB are loaded with the new values passed.

In either case, the timer is enqueued in the timer queue, tq, based upon its expiration time (i.e. etime + delay). Its computed differential count is stored in its TMRCB, and it is singly-linked into tq. Then its handle is loaded into the location at tmrhp.

The address of the user's timer handle variable is saved in the TMRCB so the timer handle can be cleared when the timer stops or is stopped. This is necessary to avoid an aliasing problem for one-shot timers. If not done, a timer could time out before it is accessed again. This would release the TMRCB which could then be re-used for a new timer. Then, a subsequent operation for the old timer would operate on the new timer — not what was intended.

When the timer times out, lsr is invoked with parameter stored in the TMRCB. This is set to 0, but can be changed by smx_TimerSetLSR(tmr, lsr, opt, par).

smx_TimerStartAbs() is identical to smx_TimerStart() except that it accepts an absolute time from system start (i.e. etime == 0), rather than a delay. This is useful to ensure that correct timing relationships are maintained between multiple timers. If delays were used, a tick might occur between timer starts, resulting in timers not being synchronized, as expected. See the smx User's Guide, Timers chapter for an example of using absolute timer starts.

Notes

1. Do not declare a timer handle as an auto variable. When the timer times out or is stopped, the timer handle location will be cleared. This will cause an error if the function that started the timer has returned and this location is being used by another function.
2. Failure to restart a running timer, due to an error, does not stop it.

Examples

```
LCB_PTR  lsra;
TMRCB_PTR  tmra;

smx_TimerStart(&tmra, 10, 0, lsra, "tmra");

void lsra_main(u32 par)
{
      /* perform timeout function */
}
```

The above example shows creating a one-shot timer that invokes lsra to perform a timeout function after 10 ticks. This occurs only once, and tmra deletes itself.

```
smx_TimerStart(&tmra, 10, 10, lsra, "tmra");
```

This creates a cyclic timer which does the same after 10 ticks, then every 10 ticks, thereafter, until it is stopped.

smx_TimerStop

BOOLEAN smx_TimerStop (TMRCB_PTR tmr, u32* tlp=NULL)

Type SSR

Summary Stops timer, loads its time left into location tlp, and deletes timer.

Compl smx_TimerStart(), smx_TimerStartAbs()

Parameters

tmr	Timer to stop.
tlp	Pointer to location to store time left.

Returns

TRUE	Timer stopped or was already stopped.
FALSE	Timer not stopped due to error.

Errors SMXE_INV_TMRCB Invalid timer handle.

Descr Removes timer from the timer queue, tq. Its differential count is added to that of the next timer, if any. The total time remaining for timer is computed and loaded into the location pointed to by tlp, unless tlp is NULL. The timer's TMRCB is cleared and returned to the timer pool and *(tmr->tmhp) = NULL, so that the tmr cannot be accessed again.

If tmr == NULL, 0 is loaded into *tlp, and TRUE is returned. The condition occurs if attempting to stop a timer that has already been stopped, or never started.

Note Do not create a derivative timer handle because it will not be automatically cleared, which can cause an aliasing problem — see discussion in smx_TimerStart()

Example

```
TMRCB_PTR  tmra;
u32  time_left;

smx_TimerStop(tmra, &time_left);
```

tmra is stopped, and the time left for the timer is stored in time_left.

smx Utility Functions

smx_ConvMsecToTicks

u32 smx_ConvMsecToTicks (u32 msec) rounded up
u32 smx_ConvMsecToTicksRound (u32 msec) rounded to nearest value

Type Unrestricted macros

Parameters msec Time in milliseconds to convert.

Returns time in ticks

Descr Converts milliseconds into ticks, rounded up to the next tick or rounded to the nearest tick, respectively, where the tick rate specified by SMX_CFG_TICKS_PER_SEC in acfg.h. The precision of the conversion depends on the tick rate.

Example

```
u32 uticks = smx_ConvMsecToTicks(24);
u32 rticks = smx_ConvMsecToTicksRound(24);
```

For a tick rate of 100Hz, uticks = 3, rticks = 2.

smx_ConvTicksToMsec

u32 smx_ConvTicksToMsec (u32 ticks) rounded up
u32 smx_ConvTicksToMsecRound (u32 ticks) rounded to nearest value

Type Unrestricted macro

Parameters ticks Time in ticks to convert.

Returns time in milliseconds

Descr Converts ticks into milliseconds, rounded up to the next millisecond or rounded to the nearest millisecond, respectively. This macro is intended to convert a small number of ticks, not a large number such as smx_etime, which could cause overflow.

Example

```
u32 umsec = smx_ConvTicksToMsec(9);
u32 rmsec = smx_ConvTicksToMsecRound(9);
```

For a tick rate of 100Hz, umsec = 90, rmsec = 90.

smx_DelayMsec

BOOLEAN smx_DelayMsec (u32 msec)

Type Macro that maps to SSR

Parameters msec Time to delay, in milliseconds.

Returns TRUE Delay completed.
FALSE Error.

Descr Delays for at least the specified number of milliseconds, as close as possible, to the precision of a tick, which may be many milliseconds, depending on the tick rate. Uses smx_ConvMsecToTicks() to convert to ticks, then calls smx_TaskSuspend(SMX_CT, ticks+1) to delay the current task. Adds 1 tick so the delay is at least as long as intended, since the next tick may be just about to occur. May be used only from tasks, not ISRs since it calls an SSR, and not from LSRs since it waits. During the delay the same and lower priority tasks can run.

Example

```
smx_DelayMsec(5);       /* wait 5 milliseconds */
```

For a 100 Hz tick rate, this would delay between 10 and 20 milliseconds, since 1 tick is the minimum delay.

smx_DelayTicks

BOOLEAN smx_DelayTicks (u32 ticks)

Type Macro that maps to SSR

Parameters ticks Time to delay, in ticks.

Returns TRUE Delay completed.
FALSE Error.

Descr Delays for the specified number of ticks, using smx_TaskSuspend(SMX_CT, ticks + 1). Adds 1 tick so the delay is at least as long as intended, since the next tick may be just about to occur.

Example

```
smx_DelayTicks(2);
```

For a 100 Hz tick rate, this would delay between 2 and 3 ticks.

smx_ERROR

void smx_ERROR (SMX_ERRNO errnum, void* handle)

Type Macro calling smx_EM()

Summary This is the smx error service macro. It switches to the system stack or main stack, then calls the smx error manager, smx_EM(). smx_EM() saves errnum in smx_errno and in smx_ct->err or in smx_clsr->err; it increments smx_errctr and smx_errctrs[errnum]; it makes entries in EB and EVB; and it displays an error message. It then calls smx_EMHook() to allow application-specific error handling.

If a stack overflow has occurred, subsequent stack overflow reporting is inhibited, the stack overflow callback function is called, if implemented, and the task is allowed to continue if the

stack pad has not been exceeded. If an irrecoverable error has occurred, smx_EMExitHook() is called to allow application code to recover.

smx_srnest must be > 0 when smx_ERROR() is called, in order to avoid reentry due to an interrupt. Also, it may not be called from an ISR, for the same reason. Interrupts are enabled during execution of smx_EM(). The related macros, smx_ERROR_EXIT() and smx_ERROR_RET(), are used within SSRs to call smx_ERROR().

smx_EBDisplay

void smx_EBDisplay (void)

Type Bare function

Summary Displays all entries in EB from start to end in the left panel of the display. Will scroll from bottom to top if EB has more records than there are display lines on the screen. If SMX_CFG_ERROR_MSGS is true, shows full error messages, else just error numbers. Should be called only from a low-priority task because it polls the UART to send characters.

smx_Go

void smx_Go (void)

Type Function

Summary Initializes smx from information in acfg.h

Parameters none

Returns none

Errors SMXE_INSUFF_HEAP
SMXE_SMX_INIT_FAIL

Descr smx_Go() initializes the error manager, event buffer, ready queue, timer queue, and creates the LSR queue, task timeout array, smx LSRs, smx_Idle, stack pool, and other smx objects. It then starts smx_Idle with ainit() as its main function and begins operation in the task environment. smx_Idle runs at maximum priority, PRI_SYS, until ainit() finishes. The above are the dominant errors; other errors may be reported. This function is intended to be called only once, from main().

smx_Go() uses constants in acfg.h. These control the amounts of memory used by smx objects, as well as the tick rate, stack parameters, and other smx features.

Example

```
void main(u32)
{
    sb_IRQsMask();
    smx_Go();
}
```

It is important to mask interrupts (not just disable) until ainit() begins running.

smx Glossary

This glossary defines smx terminology used in the manuals. Terms are in alphabetical order. The smx_ or SMX_ prefix is generally omitted (else nearly all entries would be under “s”). So, for example, look for “ct”, instead of “smx_ct”. An exception is that errors are listed under “SMXE” in order to keep them together.

access conflict Occurs when two routines try to simultaneously access a non-sharable system resource. Access conflicts due to preemption are similar to those caused by hardware interrupts. See also: critical section and lock.

active task A task which is running or suspended, but not stopped.

adjusted size of a block allocated from the heap is the next larger multiple of 8 if the requested size is not a multiple of 8. This is the size that is actually requested.

allocation policy as applied to the heap, means specifying how a best-fit chunk is found and also specifying the minimum remnant size for splitting a new chunk from a larger chunk that has been found. The allocation policy effects performance vs. memory efficiency.

atomic As applied to code, means that a group of statements cannot be interrupted by other code.

automatic merge For eheap and smx heap means that chunk merging is automatically controlled.

autostop Running through the last brace of a task’s main function results in the task automatically stopping. When this occurs, the task can be restarted by another task.

background In an smx system, tasks are considered to be in the background, and ISRs and LSRs are considered to be in the foreground.

bare block is a data block which is not linked to an smx control block. Examples are base blocks, DAR blocks, heap blocks, and static blocks.

bare function An ordinary C function that is part of the smx or smxBase API and is prefixed with smx_ or sb_. “bare” emphasizes that the function is not task-safe and care should be exercised if called from a task. Normally used in ISRs, LSRs, and SSRs.

bare macro An ordinary C macro similar to a bare function.

base block is a data block from an smxBase block pool. It is obtained with sb_BlockGet().

base block pool is an smxBase data block pool created by sb_BlockPoolCreate(). A base pool is controlled by a pool control block of type PCB, which is identical to an smx PCB, but statically defined. See smxBase User’s Guide for more information.

BCB **block control block**. An smx block consists of a BCB linked to a bare block. BCBs come from the smx_bcb pool. See xtypes.h.

BCB pool consists of a singly-linked list of free BCBs pointed to by smx_bcbs.pn. The link pointer is in the first word of each free BCB, and the last free BCB has a NULL link.

BCB_PTR smx block handle type

best-fit chunk The chunk in a large heap bin, which is the smallest chunk that is big enough to satisfy an allocation request. If the bin is sorted by increasing size, this will be the first large-enough chunk found in the bin.

bfp **bin fix pointer (eh_hvp[hn]->bfp)** is used by smx_HeapBinScan() to point to the starting chunk for the next backward scan to fix a heap break.

bin See heap bin**.**

bin leak occurs when cmerge is ON and chunks freed are merged with adjacent free chunks and the resulting larger free chunks are moved to larger bins.

bin-type heap A heap that uses bins to "store" free chunks. Chunks are not actually moved from the heap to the bins. Rather they are linked into the bins. Each bin stores one or more chunk sizes.

binary semaphore has only two states: 0 and 1. smx_SemSignal() puts it into the 1 state if no tasks are waiting; smx_SemTest() puts it into the 0 state. Once in the 1 state, additional signals have no effect; once in the 0 state, additional tests suspend tasks having timeouts.

block is two or more adjacent memory locations, excluding standard data types..

block migration refers to the process of making a block into a message to pass it to the background and unmaking a message into a block to pass it to the foreground. See smx User's Guide, Exchange Messaging chapter.

block pool A pool of equal-size blocks controlled by a Pool Control Block (PCB). See base block pool and smx block pool.

bmap **bin map** has one bit per bin. If the bit is set, the bin contains one or more chunks.

BOOLEAN A TRUE(1)/FALSE(0) variable. This is the traditional C definition. To enhance reliability, we recommend that you test for TRUE as !0 rather than 1.

bound A bound stack is a permanent stack that is released only if the task is deleted. In this case task->flags.stk_perm is set. Bound stacks are either preallocated or allocated from a heap when a task is created. In the latter case the size is specified in smx_TaskCreate().

BPCB **block pool control block** controls an eheap block pool. It has the number of blocks in the pool, number inuse, maximum number inuse, pointers to the first and last blocks of the pool, and the free block list pointer. See eheap.h.

bridge is formed when heap links cannot be fixed by smx_HeapScan(). When this happens, the chunk with a broken forward link is linked to the chunk with a broken backward link. Thus chunks in between are bridged over. SMXE_HEAP_BRKN is reported. Bridging is intended as a temporary measure to allow a controlled system shut down.

broadcasting is accomplished by sending a message to a broadcast exchange. All tasks receiving from the exchange will receive the message handle and the message block pointer, but not the message block. The message block pointer allows a task to read the message. It may also be used by a task to write its section of a message. This is called *distributed message assembly*.

bs_fwd mode causes forward heap scans which is the method used to check for breaks.

bsmap **bin sort map** has one bit per heap bin. If the bit is set, the bin needs to be sorted.

bsp **bin scan pointer (eh_hvp[hn]->bsp)** is used by smx_HeapBinScan() to point to the starting chunk for the next forward scan to check for heap breaks.

callback function A function called by the scheduler or certain smx services, which allows the user to add custom operations. See descriptions in this manual for smx_EventFlagsSet(),

smx_EventGroupSet(), smx_MsgSend(), smx_PipeGetPktWait(), smx_PipePutPktWaitStop(), smx_SemSet(), smx_SemSignal(), smx_TaskDelete(), and smx_TaskStart(). See smx User's Guide, Tasks chapter, task callback functions section for details concerning task callbacks.

CB **control block.** A structure that stores control information for a system object. Each object type (e.g. task, semaphore, or message) requires a different control block format since different control information is needed for each. There is a pool for each type of control block. For example, smx_mcbs, contains message control blocks. The size of a pool is determined by a configuration constant in acfg.h, for example SMX_CFG_NUM_MSGS. See xtypes.h, bdef.h, and eheap.h for control block formats.

cbtype **control block type**. This field is present in nearly all control blocks, and it is always in the same position, if present. Values are defined in xdef.h, for example SMX_CB_TASK.

CCB **chunk free control block** is placed at the start of a free chunk. It provides information necessary to manage the free chunk. A CCB contains 24 bytes.

CDCB **chunk debug control block** is placed at the start of a debug chunk. It provides information necessary to debug heap problems. In addition to fl and blf, it has chunk size, time of allocation, owner, and one fence ahead of the data block. A CDCB contains 24 bytes.

CICB **chunk inuse control block** is placed at the start of an inuse chunk. It provides forward and backward links for the heap. A CICB contains 8 bytes.

ceiling See **priority ceiling.**

CHK_OVH **chunk overhead** consists of the metadata in a chunk which is necessary to manage it. The size of an allocated chunk at least = block_size + EH_CHK_OVH.

chunk A block of memory used by the heap. A chunk consists of a chunk control block (CCB) used by the heap code and a data block used by the application. A chunk is thus larger than the data block, which it contains. The smx heap supports three types of chunks: free, inuse, and debug.

client task is provided a service by a *server task*. Typically, a client task sends a message to a message exchange, then waits for a response.

clsr **smx_clsr** is the handle of the currently running LSR. If NULL, no LSR is running. While an LSR is running, the current task is blocked from running.

cmerge mode for heap n is controlled by the hvn.mode.fl.cmerge flag, where hvn is the eheap variable structure, EHV, for heap n. It controls whether freed chunks are merged. When ON, freed chunks are merged with adjacent free chunks to avoid allocation failures by reducing fragmentation. When OFF chunk merging is inhibited, which helps to build and maintain bin populations. Can be turned ON or OFF via smx_HeapSet().

complementary call The smx call that performs the inverse operation of a particular call. For example: smx_MsgSend() vs. smx_MsgReceive().

complementary pipe function An smx pipe function that may be used at the other end of the same pipe. Most combinations of PipePut functions and PipeGet functions are permitted.

context switch When a task switch occurs, the context of one task is replaced with the context of another task. Typically a task context consists of the processor registers.

control block See CB.

control block pool smx control blocks are grouped into pools controlled by pool control blocks (PCBs). For example, the TCB pool is controlled by smx_tcbs. Control block pools and their PCBs are statically allocated.

counting semaphore is the same as a **resource semaphore**.

critical section A section of code which modifies shared data or which accesses a shared system resource. Critical sections must be protected from interrupts and task preemptions.

ct **smx_ct** is the currently running task.

current chunk is the chunk that is currently being processed.

current delay For a one-shot timer, the current delay is its initial delay. For cyclic and pulse timers, the current delay is the initial delay until the first period starts. Then, for a cyclic timer, the current delay is its period and for a pulse timer, the current delay is the delay until the end of the current HI or LO period.

DAR **dynamically allocated region** — A region of memory for dynamically allocated blocks. A DAR is a primitive heap that allows allocating blocks but not freeing them (except the last allocated). DARs are no longer used in smx but may be useful for other purposes. See smxBase User's Guide for DAR discussion.

data block is a block intended to hold data, as distinct from a control block, which holds control information.

dc See **donor chunk.**

DCB_PTR **DAR control block pointer** points to a structure containing the pointers for a DAR. See bdef.h for definition. A DAR control block is initialized by sb_DARInit().

deadline is the time when a task must complete an operation or a failure may occur.

deadlock or deadly embrace occurs when two tasks are waiting upon resources owned by the other. As a consequence, neither can complete. To avoid deadlocks, tasks should get resources in the same order and release them in the reverse order or use mutexes with ceiling priority.

debug chunk is an inuse heap chunk that contains a Chunk Debug Control Block, CDCB, and heap fences around the data block. The number of fences is user-specified. The CDCB has several more fields than the CICB to aid debugging heap problems and for heap monitoring. See **debug mode** about controlling whether allocations become debug chunks.

debug mode for heap n is controlled by the hvn.mode.fl.debug flag, where hvn is the eheap variable structure, EHV, for heap n. When ON, allocations produce debug chunks; when OFF, allocations produce inuse chunks. It starts OFF and can be turned ON or OFF via smx_HeapSet().

debug version The version of smx, middleware, or application intended for debugging. It is compiled with no optimization, debug symbolics enabled, and SMX_BT_DEBUG defined. The latter is used to enable alternative debug code for smx, such as putting tables into RAM instead of ROM.

dequeue The process of removing a task or a message control block from a queue. Dequeueing is a logical process. Control blocks are not moved — all stay in the same physical location.

When a queue becomes empty or an object is not in a queue, its fl == NULL. bl is not changed, for efficiency.

dispatch Dispatching a task is the process of starting it to run. This is done by the task scheduler. The task dispatched is the top task, unless the current task is locked, in which case it is the one dispatched.

distributed message assembly is where components of a message (e.g. header and payload) are assembled by different tasks, which have each received a pointer to the message block from a broadcast exchange or a proxy message. See smx User's Guide, Exchange Messaging chapter, broadcasting messages and following sections.

donor chunk is located between the lower heap and the upper heap. It supplies small chunks for the lower heap. If the small bin array bin for the desired size is empty, the chunk is taken from dc. This helps to separate small chunks from large chunks in order to reduce fragmentation.

dormant A task is dormant if it is stopped with infinite timeout. Such a task will not run again unless it is started by another task.

double free occurs when smx_HeapFree() attempts to free a chunk that has already been freed. If the chunk has not already been reallocated or merged, this is detected and SMXE_HEAP_ERROR reported.

dynamic A dynamic object can be created and deleted at run time. All smx objects can be dynamically created and deleted.

dynamically allocated region See DAR.

EB See **error buffer**.

ec See **end chunk.**

EG **event group** consists of 16 event flags in an event group control block, EGCB. The flags can be set, reset, and tested by event flag service calls. The AND, OR, or AND/OR combinations of the event flags can be tested. Multiple tasks can wait an event group, each for its own combination of flags.

EGCB **event group control block** controls an event group. It contains forward and backward links for the task wait queue, flags, and other fields. See xtypes.h.

EGCB pool consists of a singly-linked list of free EGCBs pointed to by smx_egcbs.pn. The link pointer is in the first word of each free EGCB, and the last free EGCB has a NULL link.

EGCB_PTR event group handle type.

eheap **embedded heap.** RTOS-agnostic heap developed for embedded systems. smx heap is based upon it. See eheap User's Guide for more information.

EM **smx_EM()**. See Error Manager.

EMHook smx_EMHook() is a callback function from smx_EM() for the application to add error processing code for non-catastrophic errors.

end chunk is the last chunk of a heap. It is an 8-byte, inuse chunk with no data block. hvn.px points to it.

enqueue The process of putting a task or a message into a queue. This is done by changing forward and backward links (fl and bl) in appropriate control blocks to add the new item to the queue. Most queues are priority queues, in which case it is necessary to search in order to place the task or message after the last object of equal priority. Some queues are FIFO queues for which the new object is placed at the end of the list. The ready queue, smx_rq, is a layered priority queue.. Enqueueing is a logical process. Control blocks are not moved; all stay in the same physical location.

EQCB **event queue control block** controls an event queue. It contains forward and backward links for the task wait queue, the event queue name, and other fields.

EQCB pool consists of a singly-linked list of free EQCBs pointed to by smx_eqcbs.pn. The link pointer is in the first word of each free EQCB and the last free EQCB has a NULL link.

EQCB_PTR event queue handle type.

EREC **error record format** for the error buffer (EB) contains fields for etime, error number, and a handle identifying the source of the error. See xtypes.h.

err **smx_ct->err** is the last error made by this task. If err == SMXE_TMO, a timeout has occurred; otherwise an error has occurred. In either case, the return value is not valid.

errctr **smx_errctr** counts all smx errors since system startup.

sb_errctr counts all smxBase errors since system startup.

errctrs **smx error counters**, smx_errctrs[]. Contains a one byte counter for each smx error type. Accessed using smx_errno as the index. Compare the sum of all counters to smx_errctr to determine if any have overflowed.

errno **smx_errno s**tores the error number of the last error detected by smx. **task->err** is the last error caused by a particular task.

sb_errno stores the error number of the last error detected by smxBase. Some smx services use smxBase services, so an smxBase service may be the actual cause of a failure.

error buffer **smx_EB** is an array of error records stored cyclically — the oldest is overwritten by the newest. Space for EB is allocated in the linker command file. It is initialized and cleared by smx_EMInit().

error manager **smx_EM()** is the smx error manager. It is called whenever an error is detected by smx. It updates the err globals below and smx_ct->err. It saves information in EB and logs the error in EVB, if error logging is enabled by SMX_EVB_EN_ERR in xevb.h. An error indication is displayed on the console. This can be an error message, such as "smx MTX ALRDY FREE" if SMX_CFG_ERROR_MSGS in xcfg.h is set or just an error number, if not. The callback function, smx_EMHook() is called to permit error processing. For most errors, control then goes back to the point of call with a failure return value. However, if an irrecoverable error has occurred, e.g. SMXE_HEAP_BRKN, the callback function, smx_EMExitHook(), is called to recover or to reboot the system.

sb_EM is the smxBase error manager. It is called whenever an error is detected by smxBase. It sets sb_errno = error, increments sb_errctr, and displays an error message on the console.

error type **smx error types** are defined in xdef.h. An enum is used for compilers which permit byte enums; defines are used for other compilers. There are 85 smx error types.

smxBase error types are defined in bdef.h. There are 16 smxBase error types.

etime **elapsed time** in ticks since the last reset. 31 bits. For 100 ticks per second, allows 248 days of elapsed time. Used for timeouts and waits. Stored in smx_etime.

etime rollover Occurs when etime and all active timeouts are >= 2^31. When a rollover occurs, the top bit of etime and all active timeouts is cleared. This is performed in the idle task, with LSRs disabled so that smx_KeepTimeLSR and smx_TimeoutLSR, which perform all timing functions, cannot run.

EVB

event buffer **smx_EVB** logs system events, such as task switches, LSR runs, ISR runs, SSR calls, and user events. Each record starts with a start-of-record marker, 0x5555rrss, where rr = record type and ss = record size in words. For example, 0x55550304 is the ISR start record (see record types in xevb.h). All records include a precision timestamp and other fields such as the current ISR, LSR, or task handle, user parameters, etime, error number, and SSR id and parameters. This information is analyzed by smxAware to display an event log and graphical event timelines.

event flag An event group has 16 event flags, each of which indicates the occurrence of one event. Setting an event flag may cause a match and result in one or more tasks being resumed.

event queue An event queue permits a task to be resumed or restarted after a specified number of events have occurred while it is waiting. Tasks are enqueued, in order, by their differential counts so only the counter in the first task need be decremented.

exchange A **message exchange** is an smx object, which permits messages to be exchanged between tasks. It is defined by an exchange control block, XCB. Exchanges have three modes of operation:

SMX_XCHG_NORM	Normal exchange.
SMX_XCHG_PASS	Pass exchange.
SMX_XCHG_BCST	Broadcast exchange.

See descriptions of each type, below, and see smx User's Guide. Exchange Messaging chapter for more information.

external fragmentation refers to wasted space in a heap due to free blocks being more numerous and smaller than is useful and separated by inuse blocks so they cannot be merged.

FALSE 0 or !TRUE.

fas SecureSMX. First active slot in an MPU above static slots, if any.

fence is a word containing EH_FENCE_FILL defined in eheap.h. It can be any pattern as long as bits 1 and 0 are 1's. Fences surround the data block in debug chunks to permit small data block overflows without damaging the heap.

fill mode is controlled by the **hvn**.**mode.fl.fill** flag in the heap n variable structure. It can be turned ON or OFF by smx_HeapSet(). When ON, all blocks freed or allocated, dc, tc, and new fences are filled with unique patterns. When OFF, fills do not occur.

flyback There are two LSR flybacks implemented in the scheduler: start flyback and resume flyback. Since the scheduler runs almost completely with interrupts enabled, just before starting or resuming a task, it checks if any LSRs are ready to run. If so, it runs them, then flies back to check if a higher priority task has become ready due to the LSRs. This is done to minimize LSR and task latencies. The SVC handler has an LSR flyback at the end.

foreground In an smx system, ISRs and LSRs are considered to be in the foreground, and tasks are considered to be in the background.

foreign stack A foreign stack is a non-smx stack. Some third-party library routines switch to their own stacks. This is especially likely if they are called via a software interrupt. Stack checking must be turned off while using a foreign stack. This can be done with smx_TaskSet(task, SMX_ST_STK_CHK, 0) to turn off, and with 1 to turn on.

fragmentation See **external fragmentation.** There also is **internal fragmentation,** which is wasted spare space inside of heap chunks.

frame A unit of time for capturing profile information, specified in ticks by SMX_CFG_RTC_FRAME in acfg.h. SecureSMX also has a runtime limit frame. See the SecureSMX User's Guide, Runtime Limiting chapter.

free() Generic heap free operation that frees inuse chunks to the heap.

free chunk A heap chunk that is not in use and thus free to be allocated. A free chunk consists of a 24-byte Chunk Control Block, CCB, and free space.

free chunk list Doubly-linked list of free chunks in a heap bin. Free forward links (ffl's) and free backward links (fbl's) in the bin and in each chunk are used to create the list. All chunks in the list are of the correct size for the bin.

gate semaphore resumes all waiting tasks with one signal.

handle A handle is a location in memory that contains the address of a control block. Hence, a *handle* is a special type of pointer that points to the control block of an smx object. Different smx objects have different handle types. For example, a task has a TCB_PTR handle type. Handles are used to access and control smx objects.

handler mode **hmode** is one of three Cortex-M modes of operation. This mode is privileged and uses the main stack, also referred to as the **system stack**.

handle table **smx_ht** allows assigning names to non-smx objects. Handles are added by smx_HT_ADD(), and they are removed by smx_HT_DELETE(). The handle table is used by smxAware.

hard real-time means that a system failure may occur if a deadline is not met.

heap A heap is a region of memory from which variable-size blocks can be dynamically allocated and to which they can be dynamically freed, when no longer needed.

heap bin A heap bin heads a free list of doubly-linked chunks of a certain size or small range of sizes. Chunks are freed to bins and allocated from bins, when possible. This results in faster allocations than searching the heap for best-fit chunks.

heap block is a data block allocated from the heap that is contained within a chunk.

heap failure Inability for the heap to supply a desired size block. Usually caused by excessive fragmentation. This is indicated by the SMXE_INSUFF_HEAP error.

heap range test is a test of a chunk pointer to verify that it is within the range of the selected heap. smx range tests all chunk pointers, before use.

heap stack is a stack allocated from the heap. A heap stack is permanently bound to a task and remains bound until the task is deleted.

hfp **heap fix pointer**, hvn.hfp, points to the starting chunk for the next smx_HeapScan() backward run.

hhwm **heap high-water mark**, hvn.hhwm, is the largest value of hused since the heap was last initialized.

high-water mark is the maximum number of bytes of stack or heap used since initialization. Each task stack high-water mark is saved in task->shwm. The system stack high-water mark is saved in smx_sshwm. The heap high-water mark is saved in hvn.hhwm. These values are displayed in smxAware and can be used to tune stack and heap sizes.

hookd flag task->flags.hookd == 1 enables task->:

1. cbfun(SMX_CBF_EXIT) to be called by the scheduler to preserve an extended task state on suspend.
2. cbfun(SMX_CBF_ENTER) to restore the extended task state on resume.
3. cbfun(SMX_CBF_STOP) to do operations on task stop, and
4. cbfun(SMX_CBF_START) to do operations on task start.

See smx_TaskSet() in this manual and the task callback function section of the Tasks chapter in the smx User's Guide for more information.

host system Refers to the development system on which application software is edited, compiled, and linked and which runs the debugger

hs_fwd mode **heap scan forward mode** is controlled by the **hvn.mode.fl.hs_fwd** flag. It starts ON and controls the direction of heap scans. It is an internal mode, not user controlled.

hsp **heap scan pointer** points to the starting chunk for the next smx_HeapScan() forward run

ht See **handle table**.

hused **heap used** is the total heap space currently allocated, including chunk overhead.

idleup **smx_idleup** indicates that the idle task has been temporarily boosted to a higher priority in order to complete scanning a stack in the scanstack pool and moving it to the freestack pool so that a waiting unbound task can run.

init mode for a heap is controlled by the **hvn.mode.fl.init** flag. It starts OFF and is set ON when the heap has been initialized. It can be turned ON or OFF by smx_HeapSet(). It must be turned OFF to reinitialize the heap.

internal fragmentation In a heap, it refers to spare space in a chunk due to it being larger than necessary for the block it contains. In a block pool, it refers to wasted space due to blocks being larger than usually necessary and to block pools containing more blocks than usually necessary.

interrupt An action which interrupts program execution by means of the processor's interrupt mechanism. Also called a hardware interrupt. Interrupts cause Interrupt Service Routines (ISRs) to run.

interrupt latency is the time from the occurrence of an interrupt until the ISR to process it starts running. Interrupt latency = processor latency + smx latency + application latency. The latter two are caused by disabling interrupts for critical sections of code. smx does not disable interrupts

in LSRs and SSRs, and only briefly in the scheduler and other places. smx interrupt latency is comparable to processor latency.

interrupt service routine See **ISR**.

inuse chunk A heap chunk which is currently being used. It contains the 8-byte chunk inuse control block, CICB, the data block being used by the application, and spare space.

ISR **interrupt service routine**. A function which handles a hardware interrupt. An ISR is usually invoked via a vector stored in an interrupt vector table (IVT), however various mechanisms are used by different processors. An **smx ISR** is one that may invoke an LSR. As a consequence, it must start with smx_ISR_ENTER() and end with smx_ISR_EXIT(). **Non-smx ISRs** are free of this requirement as long as they have higher priority than any smx ISR or do not enable interrupts. ISRs cannot call smx services other than smx_LSR_INVOKE() and bare pipe functions. See smx User's Guide, Service Routines chapter for more information.

large bin A heap bin that stores a range of chunk sizes.

last turtle is the last chunk in a large heap bin free list that might be smaller than a chunk before it. It is called a *turtle* because it moves forward very slowly in a bubble sort.

limited SSR An smx service that can only be called from tasks and not from LSRs. These are primarily SSRs that stop the current task, such as smx_MsgReceiveStop().

linear heap A heap that must be searched sequentially to find a large-enough chunk to allocate.

link service routine See **LSR**.

localization As applied to heaps, means if chunks being allocated and freed during a short period of time are physically close, cache hits will increase.

locked A task is locked if smx_lockctr > 0. When locked, a task cannot be preempted. However ISRs and LSRs can run.

logical structure A heap structure that provides a more efficient means of searching for block allocations than the physical structure. eheap provides an array of heap bins for this purpose.

lq **LSR queue** is a cyclic queue, which contains the handle and parameter of each invoked LSR that is waiting to run, in the order that it was invoked. If lq overflows due to a newly invoked LSR overwriting an LSR that has not yet executed, SMXE_LQ_OVFL is reported. In this case, SMX_CFG_LQ_SIZE in acfg.h should be increased.

LSR **link service routines** perform deferred interrupt processing and call system services, which ISRs cannot do. LSRs are normally invoked from ISRs, although they can be invoked from tasks or LSRs. An LSR is passed a 32-bit parameter each time it is invoked. Unlike tasks, the same LSR can be invoked multiple times, usually with a different parameter each time. Once all ISRs are done, LSRs execute in the order they were invoked. This is helpful to handle bursts of interrupts.

void lsr_main(u32 par) is the standard LSR function format. LSRs never return any value. The LSR parameter can be defined as a different type:

```
void lsra_main(MCB_PTR msg);
```

but when invoked msg must be typecast:

```
smx_LSR_INVOKE(lsra, (u32)msg);
```

There are three types of LSRs:

1. Trusted LSRs.
2. Safe LSRs (pmode).
3. Safe LSRs (umode).

Trusted LSRs are the type used in smx. They run in handler mode, like ISRs. Safe LSRs are available only with SecureSMX. See smx User's Guide, Service Routines chapter for more information.

macro A set of statements inserted in place of an identifier, by the compiler or assembler preprocessor. smx macro names are all caps, except for the smx_ prefix, so they can be distinguished from functions. smx constants are all caps, including the SMX_ prefix to distinguish them from smx macros. Most smx macros are also available as functions to save memory. SMX_CFG_MACROS in xcfg.h controls which are selected.

main() Application entry point for C/C++ programs, called by startup code. See startup for more information.

main function The main function of a task is the function which the task scheduler calls when it starts the task. Its address is stored in task->fun. The main function of an LSR is the function which the LSR scheduler calls when it starts the LSR. Its address is stored in lsr->fun.

malloc() Generic name for block allocation service of a heap.

master task A task which sends messages to a broadcast exchange. The master task retains control of the message and can release it or send it elsewhere. See smx User's Guide, Exchange Messaging chapter, broadcasting messages section.

MCB **message control block.** Each active smx message has an MCB, which contains message parameters used by smx. These include its forward and backward links for enqueueing, priority, reply index, data block pointer, block pool, and owner.

MCB pool All MCBs are in a pool, which is controlled by the smx_mcbs pool control block. The singly-linked list of free MCBs is pointed to by smx_mcbs.pn. The link pointer is in the first word of each free MCB. The last free MCB has a NULL link.

MCB_PTR Message handle type.

memory leak is loss of usable memory. This normally occurs in a heap due to failure to free blocks when no longer needed and subsequently allocating them again, which results in steady loss of free heap space. A typical example is a function that allocates a block at the start and frees it at the end, but an early exit point is added that neglects to free the block. Debug chunks help to identify leaked blocks by recording time of allocation and owner.

message An smx message consists of a data block and a message control block, MCB, linked together. Messages are identified by their handles, which are MCB pointers. They are sent between tasks and LSRs via exchanges.

message queue of other kernels is the same as the smx pipe used for intertask communication.

MIN_FRAG Configuration constant in eheap.h that defines the minimum fragment (remnant) that can be split off of a larger chunk during an allocation. This should be at least as large as the minimum chunk size that an application needs, in order to prevent accumulation of unusable small chunks.

mode flag In an event group, a mode flag represents a mode of operation, such as startup. Generally it is not desirable to clear mode flags when a match occurs.

MUCB **mutex control block**. Each active mutex has an MUCB, which contains mutex parameters used by smx to control the mutex. It has forward and backward links for a task queue, priority inheritance flag, priority ceiling, owner, next mutex in owned list, nesting count, and name. The owner is the task that currently owns the mutex. The mutex owned list links other mutexes together that are owned by the same task. The nesting count is incremented each time the same task gets the mutex and decremented each time the same task releases the mutex. See smx User's Guide, Mutexes chapter for more information.

MUCB pool All MUCBs are in a pool, which is controlled by the smx_mucbs pool control block. The singly-linked list of free MUCBs is pointed to by smx_mucbs.pn. The link pointer is in the first word of each free MUCB. The last free MUCB has a NULL link.

MUCB_PTR Mutex handle type.

multicasting consists of sending proxy messages to multiple exchanges. This provides more control than broadcasting. See smx User's Guide, Exchange Messaging chapter, proxy messages and multicasting section.

mutex A mutex is a "mutual exclusion" semaphore. It is used to limit access to critical sections of code and system resources that cannot be shared. A mutex has two states: free and owned. Only one task at a time can own a mutex. See MUCB.

NMI **non-maskable interrupt** cannot be inhibited by the processor's interrupt flag(s). This can cause access problems for shared resources and thus should be used with extreme caution. An smx ISR should never be hooked to a non-maskable interrupt because smx relies on disabling interrupts to protect critical sections.

non-smx ISR An ISR which does not interact with smx. If such an ISR does not enable interrupts or if it has higher priority than all smx ISRs, then there is no restriction on how it may be written. However, if neither of these conditions is met, then it must be started with smx_ISR_ENTER() and ended with smx_ISR_EXIT().

non-volatile registers are the registers that a C/C++ compiler expects to remain unchanged by a function call. When an SSR causes a task switch, smx saves these registers and restores them when the task is resumed. See also volatile registers.

normal exchange The ordinary type of exchange used to convey messages between tasks. Sending a message to a normal exchange results in it being passed to the top task waiting at that exchange. If no task is waiting, the message is enqueued at the exchange and given to the first task that receives a message from the exchange.

NULL Means a null pointer. It is preferrable to use NULL rather than 0 for pointers.

object There are three types of objects in an smx system:

1. Application objects.
2. System objects.
3. smx objects.

Application objects consist of arrays, functions, etc. which are unique to the application.

System objects consist of tasks, pools, blocks, messages, exchanges, queues, etc. created by an application.

smx objects consist of control blocks, variables, and constants used by smx to control the system. These are generally not available to the application.

one-shot task is a task which stops when done and releases its stack. Thus, one-shot tasks do not have infinite internal loops like normal tasks, but they can wait for resources in either the suspended state or the stopped state.

pass exchange is like a normal exchange, except that it passes the priority of the message to the task receiving the message, unless the message priority is 0. This allows more important messages to get expedited processing.

PCB **pool control block** controls smx and smxBase block pools. It has pointers to the first and last blocks of the pool, a free block list pointer, block size, the number of blocks in the pool, and other information. The PCB typedef is defined in bdef.h for use by smxBase block pools as well as smx block pools.

PCBs used for smx control block pools are statically defined in xglob.c. smx_CBPoolsCreate() creates the smx control block pools. This function is called by $Sub$$__call_ctors() in IAR startup code, in order to initialize the control block pools before C++ global object constructors run.

PCB pool PCBs for dynamically-created pools are in a pool controlled by smx_pcbs. The singly-linked list of free PCBs is pointed to by smx_pcbs.pn. The link pointer is in the first word of each free PCB. The last free PCB has a NULL link.

PCB_PTR Pool handle type.

permanent stack is a stack that is bound to a task when the task is created. Permanent stacks come from a heap or are preallocated. Unlike a temporary stack, a permanent stack remains bound to a task even if the task stops. It is only released when its task is deleted.

physical heap structure consists of all chunks in the heap, doubly-linked together in physical address order. Every chunk has a forward link, fl, and a backward link + flags, blf, for this purpose. The flags are SSP (bit 2), DEBUG (bit 1) and INUSE (bit 0). Adding flags to the back link is possible because all chunks are 8-byte aligned, hence address bits 0, 1, and 2 are always 0 and not needed for addressing.

PICB **pipe control block.** Each active pipe has a PICB, which contains pipe parameters used by smx to control the pipe. It contains forward and backward links for a task queue, pipe read and write pointers, pipe start and end pointers, pipe width, flags, and pipe name. The PICB is allocated and initialized when a pipe is created.

PICB pool All PICBs are in a pool, which is controlled by the smx_picbs pool control block. The singly-linked list of free PICBs is pointed to by smx_picbs.pn. The link pointer is in the first word of each free PICB. The last free PICB has a NULL link.

PICB_PTR pipe handle typedef.

pipe An smx object which permits transfer of packets or messages between tasks and between tasks and LSRs. Packet size is specified when the pipe is created and may be 1 to 255 bytes. Tasks and LSRs use put wait SSRs to put packets into pipes and get wait SSRs to get packets from them. Tasks can wait on full or empty pipes. Messages can be put to the back or to the front of a pipe. This type of pipe is also known as a "message queue".

Pipes also permit transfer of bytes or packets between ISRs and LSRs or tasks. ISRs use put8 and put packet functions to put bytes into pipes and get8 and get packet functions to get them out. Put and get functions, which are intended for ISR usage, do not wait and also cannot resume or restart a waiting task. This type of pipe is called an "IO pipe".

pmode SecureSMX. Privileged or protected mode.

pool A pool consists of contiguous blocks of equal size. smxBase provides base block pools and base blocks, which are bare blocks. smx provides smx block pools and smx blocks, which have block control blocks (BCBs) linked to bare blocks.

porting layer A set of functions, macros, and defines that allows moving software from one processor and operating system to another, leaving the bulk of the software unchanged.

postchunk is the heap chunk that follows the current chunk.

prechunk is the heap chunk that precedes the current chunk.

precise With respect to timing, means precise to a tick counter clock. The tick counter clock rate depends upon hardware and may correspond to one instruction clock time or many instruction clock times.

precise profiling See **profiling.**

preemptible An smx task is preemptible if it is not locked. Preemption of a task can only be performed by a higher priority task.

preemption is the process of one task running in place of another. The preempted task is suspended and the preempting task is started or resumed. In smx, preemption is caused by a higher-priority task becoming ready to run due to an external event, a service call from the preempted task, or a timeout. When caused by an interrupt, preemption can literally occur between any two machine instructions in the preempted task.

preemptive scheduling is one of many scheduling algorithms used by operating systems. Preemptive scheduling means that the highest priority ready task always runs, unless the current task is locked. This is the most appropriate scheduling algorithm for hard-real-time systems, and it is the main one used by smx.

priority smx task and message priorities range from 0 to 126. If the increasing priority direction is up, SMX_CFG_PRI_UP == 1, 0 is the lowest priority and PRI_NUM is the highest priority. If the increasing priority direction is down, If SMX_CFG_PRI_UP == 0, PRI_NUM is the lowest priority and 0 is the highest priority. The latter case has been adopted for porting applications from RTOSs like ThreadX. The first case is the preferred case for smx.

An enumerated data type is a good way to define priorities. For example:

```
typedef enum {PRI_MIN, PRI_LO, PRI_NORM, PRI_HI, PRI_MAX} PRIORITIES;
or
typedef enum {PRI_MAX, PRI_HI, PRI_NORM, PRI_LO, PRI_MIN} PRIORITIES;
```

does not depend upon priority direction, provides good readability, and allows a new level to be easily added (e.g. PRI_LO_LO and PRI_LO_HI around PRI_LO). If priorities are numbered, then many priorities must be changed to add an intermediate priority.

A set of macros is defined in xsmx.h to perform priority comparisons. For example, SMX_PRI_A(a, b) is true if priority a > priority b, regardless of priority direction.

priority ceiling is a priority possessed by an object. A task assumes this priority when it owns the object. smx mutexes can be assigned priority ceilings. The ceiling normally is the highest priority of any task that may own the object. Priority ceiling avoids unbounded priority inversion and also eliminates deadlocks for objects having the same ceiling.

priority direction See **priority**.

priority inheritance is the process of promoting a mutex owner's priority to that of the highest priority task waiting for the mutex. This is done to prevent unbounded priority inversion for the highest priority waiting task. smx mutexes support priority inheritance. They also implement **priority propagation** to other mutexes, and **staggered priority demotion** as a mutex is released by successive owners.

priority inversion occurs when a lower priority task keeps a higher priority task waiting for a resource. This is normal and predictable. **Unbounded priority inversion** occurs when the lower priority task is preempted by mid-priority tasks. The resulting delay of the higher priority task is then unpredictable and may cause it to miss a deadline. smx provides priority ceiling and priority inheritance for mutexes to deal with this problem.

priority promotion See priority inheritance.

priority propagation occurs when the owner of one mutex is waiting at another mutex and priority promotion occurs at the first mutex. The new priority will be propagated to the owner of the second mutex if its priority is lower. This process can continue for a string of mutexes.

processor architecture Typical processor architectures are ARM, ARM-M, ColdFire, etc. Within an architecture there are different processor families and processors. See the smx Target Guide for more information.

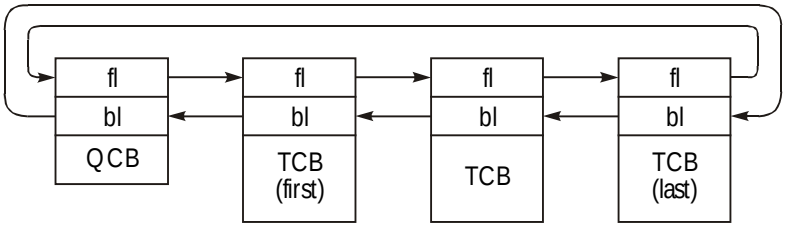
profiling smx provides precise profiling and coarse profiling. Precise profiling records run time counts (RTCs) in the TCBs of all tasks and also run time counts for all ISRs, and all LSRs. Counts are accumulated for a frame, defined as SMX_CFG_RTC_FRAME in xcfg.h, then loaded into the RTC buffer, which cyclically stores SMX_CFG_RTCB_SIZE samples for later display by smxAware or transfer to a file. smx overhead is recorded as the difference between total counts per frame and the sum of all RTCs. Coarse profiles (% idle, % work, and % overhead) are calculated from RTCs and smoothed for console display. See smx User's Guide, Precise Profiling chapter.

proxy message consists of an MCB which points to a shared message data block. A proxy message can be made from a real message with smx_MsgMake(dp, NULL), where dp points to the real message data block. Proxy messages are used for **multicasting** and **distributed message assembly**.

ptask SecureSMX. Privileged or protected task.

ptime **precise time**. This is the time derived from the input clock of the tick counter and used for profiling, time measurement, event buffer timestamps, and polling delays. It is accurate to one tick counter clock. sb_PtimeGet() is used to get ptime.

queue Most smx queues are doubly linked lists of tasks or messages as shown in the following example:



QCB is the head of the list. It represents control blocks that have queues, such as XCBs. TCBs are added or removed by changing links. For example to remove the first TCB, fl of the QCB is changed to point to the next TCB and the next TCB's bl is changed to point to the QCB. Then, the first TCB's fl is set to NULL to indicate that it is no longer in a queue. Message queues are the same, except that MCBs are linked in instead of TCBs.

ready queue holds tasks that are ready to run. It has one level per priority, starting at PRI_MIN and going up to PRI_SYS in xcfg.h. smx_rq is created by smx_Go(). The levels are in increasing priority order, which allows a level to be directly accessed by using its priority as an index. The highest priority level accepts tasks at that level and above. The lowest priority level is used by smx_Idle.

A task is enqueued in smx_rq, by indexing into it using the task's priority, then enqueueing the task at the end of the level. This is a very fast process, which is independent of the number of tasks in smx_rq. The smx_rqtop pointer is maintained in order to dequeue the top task quickly. When a task is running, smx_rqtop normally points at its TCB. When it stops running, smx_rqtop points at the next task to run.

real message is a message with both a message body and MCB as compared to a proxy message which has only an MCB.

register save area (RSA) is the area below a task stack which is used by the scheduler to save a task's non-volatile registers when it is suspended. tcb.sbp points to the start of RSA. RSA size is typically about 40 bytes, depending upon the processor. It is set by SMX_RSA_SIZE in the processor architecture header file (e.g. xarmm.h), since it is architecture dependent.

release version is the version of smx, middleware, and application intended to be embedded in the shipped product. It is compiled at high optimization, with debug symbolics disabled, and SMX_BT_RELEASE defined.

remnant The remainder of a chunk after splitting a chunk. It must be at least EH_MIN_FRAG (eheap.h) bytes or the initial chunk will not be split. It will always be above the allocated chunk and it will be merged with a free postchunk if cmerge is ON

reply smx_MsgSend() has a reply parameter. It is an XCB handle. Its index is stored in the MCB of the message being sent. This allows the receiving task to send a reply message, which is useful for client/server designs.

resource In a multitasking system, the term resource is generally used to mean something that tasks use, such as an object, data, or a peripheral. Shared resources normally are protected with mutexes or other means.

resource semaphore has an internal count corresponding to the number of resources it controls. Each smx_SemTest() decrements the counter and passes until the count reaches 0. After that, tasks must wait at the semaphore for signals indicating resources released by other tasks.

response time The time from the occurrence of an interrupt until an ISR, LSR, or task begins running to process the interrupt. ISR response time is governed by interrupt latency of the processor plus run times of higher-priority ISRs. LSR response time is the sum of all ISR run times that might occur ahead of it and of all LSR run times that may be enqueued ahead of it. Task response time is the sum of the above plus task switching time, assuming that it is the highest priority task.

restart Means that a task has been stopped and now is restarting from the beginning of its task_main(). A task restart can be due to the occurrence of the event for which the task was waiting, a timeout, or a direct start from another task or LSR.

resume When a task resumes, it continues running from where it was suspended. All registers are restored to their previous values, even though other tasks and service routines may have run in the interim. Many smx services suspend a task until a desired event occurs, then resume it. Suspended tasks can also be directly resumed by another task or LSR.

ROM version The version of an application intended to be embedded in the shipped product. It is compiled at high optimization with debug symbolics disabled and located in ROM and with SMX_BT_ROM defined. See also: release version and debug version.

round-robin scheduling means that tasks run one after the other until all have run and then the process repeats. This is normally a cooperative scheduling algorithm, in which running tasks voluntarily yield to allow the next task run. It can be accomplished by using smx_TaskBump() for a task to move itself to the end of its priority level in rq. All tasks in the round-robin group must have the same priority. Note that higher priority tasks can preempt at any time, but lower priority tasks can run only when the round-robin group stops running.

rq See **ready queue**.

rqtop **smx_rqtop** points to the top task in the ready queue. This is the first task in the top occupied level of the ready queue and normally will be the next task to run.

RSA See **Register Save Area**.

run context The run context of a task consists of the contents of all registers, the task's stack, its local variables and the information in its TCB. All of these must be preserved when a task is suspended so the task can be resumed from exactly where it left off. Volatile registers need not be saved for an SSR and are saved on the task stack for an interrupt. Nonvolatile registers are saved in the task's Register Save Area (RSA), and the task stack pointer is saved in task->sp.

If a coprocessor is present, its registers are also part of the context of any task using it. smx provides task callback function calls for suspend (EXIT case) and resume (ENTRY) case to save and restore extended contexts. See **callback functions**.

SBA See **small bin array** in eheap and smx heap.

SB_DATA_ALIGN Minimum alignment for all variables. Typically 4 bytes. See processor architecture header file (e.g. barmm.h).

sc See **start chunk.**

scan pattern is a recognizable pattern loaded into a stack for stack scanning. It is defined as SB_STK_FILL_VAL in bdef.h. We use 0x55555555, but you can use any value you wish.

SCB **semaphore control block.** Each semaphore has an SCB, which contains important semaphore parameters. These include forward and backward links for the task queue, mode, signal counter, signal limit/threshold, and more.

SCB pool **smx_scbs**, semaphore control block pool.

SCB_PTR Semaphore handle typedef.

sched **smx_sched** is an internal smx variable, which tells the scheduler what to do:

SMX_CT_STOP	Stop the current task.
SMX_CT_SUSP	Suspend the current task.
SMX_CT_TEST	Test for higher priority task to preempt.

In the first two cases, ct has already been removed from rq. This flag is set by SSRs

scheduler The smx scheduler is a preemptive scheduler. It consists of a prescheduler, LSR scheduler, and task scheduler. The prescheduler is entered from smx_SSRExit() or smx_ISR_EXIT(). It runs the LSR scheduler if smx_lqctr > 0, then runs the task scheduler if smx_sched > 0, else it continues the current task, smx_ct, running. The LSR scheduler runs LSRs in FIFO order from the LSR queue. The task scheduler starts tasks, suspends tasks, resumes tasks, and autostops tasks. The process of starting or resuming a task is called dispatching a task. The top task in the ready queue is the next task dispatched, unless smx_ct is stopped.

The scheduler runs with interrupts enabled, except briefly disabled in a few places. This necessitates flybacks to ensure that the latest LSR ready to run, runs before any task. The scheduler uses the system stack and is written in C, with a few assembly macros.

scheduling Scheduling consists of determining what to run next. LSRs take precedence over tasks. LSRs run in the order they were enqueued. Tasks are scheduled by going to the highest occupied priority level of rq (pointed to by smx_rqtop), then picking the first task in that level — the so called top task.

semaphore Semaphores are used for resource management, event signaling, and gating. smx supports six types of semaphores, each intended for a different purpose. smx_SemTest() allows a task to test if a condition is true at a semaphore. If not, the task waits at the semaphore. smx_SemSignal() allows a task or LSR to signal that a resource has been released or an event has occurred.

server LSR A server LSR is typically invoked by a task, ISR, or another LSR to access a resource. A server LSR is particularly useful to prevent access conflicts between ISRs, LSRs, and tasks in any combination. See smx User's Guide, Resource Management chapter, server LSRs section for more information.

server task A server task typically waits at an exchange for messages from clients. When it receives a message from a client, it performs the associated service, such as a file access, then sends a reply to the client. Server tasks are a good way to regulate access to resources and also to

perform lengthy functions for other tasks, such as decryption. See smx User's Guide, Resource Management chapter, server tasks section for more information.

service routine smx provides three types of service routines:

ISR	Interrupt service routine
LSR	Link service routine
SSR	System service routine

Service routines are managed by smx and tend to occur due to events.

signal is an indication that an event has occurred.

slave task A task which receives messages from a broadcast exchange. A slave task does not get ownership of a broadcast message. It gets only its handle and message block pointer. Usually slave tasks just read broadcast messages. However, they may also load sections of broadcast messages. See broadcasting.

sleep mode The mode into which the processor is put when the smx_SysPowerDown() service is called. This is processor dependent. Some processors have only one mode, others like Cortex-M have SLEEP and DEEP_SLEEP modes, and some processor have even more.

small bin A heap bin that stores a single chunk size.

small bin array (SBA) is an array of small heap bins in the bin[] array for a heap, starting at size 24 and consisting of consecutive bin sizes that are multiples of 8 (e.g. 24, 32, 40, ...) up to sba_top bin. SBA bins can be accessed very quickly by converting the desired block size to an SBA index, e.g. binno = size/8 – 3.

small chunk A small chunk is one that fits into an SBA bin.

smx block An smx block consists of a data block and a block control block, BCB, linked together. smx blocks are identified by their handles, which are BCB pointers. smx blocks are normally used in the same ways as base blocks and bare blocks. Their advantage is that BCBs contain an owner field, which can be used to ensure that all blocks owned by a task are released when it is deleted. Also an smx block handle is set to smx_nullcb when it is released or deleted. Hence smx blocks are more reliable than base and bare blocks.

smx block pool A block pool created by smx_PoolCreate(). An smx block pool is controlled by a pool control block (PCB), which is identical to a base PCB, except that smx PCBs are dynamically allocated, whereas base PCBs are statically defined.

smx call same as an smx service. Can be an SSR, function, or macro. See the smx Services section of this manual.

SMX_CFG_PRI_UP Controls whether priorities increase or decrease numerically. See priority.

SMX_CFG_TICKS_PER_SEC Ticks per second. Defined in acfg.h.

SMXE_ABORT An irrecoverable error has occurred. smx_EMExitHook() is called to shut down the system, or whatever is appropriate.

SMXE_ABORT_TASK An irrecoverable error has occurred for a task. smx_EMExitHook() is called to allow the user to stop or delete the task.

SMXE_BLK_IN_USE A block pool cannot be deleted because one or more of its blocks are still in use.

SMXE_BROKEN_Q Occurs when an invalid forward link or backward link is found while tracing a queue. smx services abort when a broken queue is found. The scheduler attempts to fix rq if it is broken.

SMXE_CLIB_ABORT A C run-time library function aborted due to an error and called abort() or exit(). smx_EMExitHook() is called.

SMXE_EXCESS_LOCKS Reported if smx_TaskLock() is called more than SMX_CFG_LOCK_NEST_LIMIT times.

SMXE_EXCESS_UNLOCKS Reported by smx_TaskUnlock() and smx_TaskUnlockQuick() if smx_lockctr is already 0. Indicates that the number of unlocks exceeds the number of locks.

SMXE_HEAP_BRKN smx_HeapScan() cannot fix the heap or smx_HeapBinScan() cannot fix a bin queue and it may be necessary to reinitialize the heap or reboot the system. This is treated as a non-recoverable error, and smx_EMExitHook() is called.

SMXE_HEAP_ERROR Indicates that a *double free* has been attempted and averted.

SMXE_HEAP_FENCE_BRKN A heap fence in a debug chunk does not match SMX_HEAP_FENCE_FILL (xcfg.h) pattern. This typically indicates a data block has overflowed.

SMXE_HEAP_FIXED smx_HeapScan() has fixed a heap problem or smx_HeapBinScan() has fixed a bin queue problem. No action is required. This notice will be logged in the event and error buffers.

SMXE_HEAP_INIT_FAILED smx_HeapInit() failed to initialize a heap. smx_EMExitHook() is called.

SMXE_HOLDING smx++**.** An smx_Msg object is already holding a message and cannot receive another one.

SMXE_HT_DUP smx_HTAdd() and smx_HT_ADD() report this if the name being added is already in the handle table. Enabled by SMX_CFG_HT_SCAN_DUP.

SMXE_HT_FULL The handle table is full. Increase SMX_CFG_HT_SIZE in acfg.h.

SMXE_INIT_MOD_FAIL smx_modules_init() has failed. This routine initializes the smx component modules (e.g. smxFS, smxUSBH, etc.).

SMXE_INSUFF_HEAP Not enough heap to allocate a block of the requested size. Increase SMX_CFG_HEAP_SPACE in acfg.h. When operating, smx_HeapExtend() can be used to extend the heap, or unneeded heap blocks can be released with eh_hvp[hn]->cmerge ON. If eh_hvp[hn]mode.fl.auto_rec is OFF, calling smx_HeapRecover() may work by merging freed blocks to create a large enough block for the allocation.

SMXE_INSUFF_UNLOCKS Reported by smx_TaskLockClear() if smx_lockctr is not 1, as expected. Indicates that the number of unlocks is less than the number of locks.

SMXE_INV_BCB smx block handle is not in the BCB range. Check if smx_BlockGet() or smx_BlockMake() failed.

SMXE_INV_CCB The chunk control block, CCB, pointed to by the chunk pointer for the block being freed has a forward link or backward link out of range. As a consequence, the free operation cannot be completed and has been aborted.

SMXE_INV_EGCB Event group handle does not point to a valid event group control block. Check if smx_EGCreate() failed.

SMXE_INV_EQCB Event queue handle does not point to a valid event queue control block. Check if smx_EventQueueCreate() failed.

SMXE_INV_FUNC SecureSMX. An attempt was made to call a function that is unavailable in the SVC call table, smx_ssrt[] in svc.c.

SMXE_INV_MCB Message handle does not point to a valid message control block Check if smx_MsgGet() or smx_MsgMake() failed.

SMXE_INV_MUCB Mutex handle does not point to a valid mutex control block Check if smx_MutexCreate() failed.

SMXE_INV_OP Invalid operation – An attempt has been made to create, get, or make an smx object that already exists. This prevents a hacker from exhausting a control block pool.

SMXE_INV_PAR An invalid parameter, not covered by other error types, has been passed to an smx call. Check parameters vs. the smx service description.

SMXE_INV_PCB Pool handle does not point to a valid pool control block Check if smx_BlockPoolCreate() failed.

SMXE_INV_PICB Pipe handle does not point to a valid pipe control block Check if smx_PipeCreate() failed.

SMXE_INV_PRI The priority passed to a system service is greater than SMX_MAX_PRI defined in xcfg.h. smx automatically adjusts such priorities down to SMX_MAX_PRI when encountered.

SMXE_INV_SCB Semaphore handle does not point to a valid semaphore control block Check if smx_SemCreate() failed.

SMXE_INV_TCB Task handle does not point to a valid task control block Check if smx_TaskCreate() failed.

SMXE_INV_XCB Exchange handle does not point to a valid exchange control block Check if smx_MsgXchgCreate() failed.

SMXE_INV_TIME Detected by smx_TaskSleep(time) or smx_TaskSleepStop(time) if the time parameter is already less than or equal to stime or if it is so large that more than (2^31 - 1) need be added to etime to convert it to a tick timeout.

SMXE_INV_TMRCB Timer handle does not point to a valid timer control block Check if smx_TimerStart() failed.

SMXE_INV_XCB Exchange handle does not point to a valid exchange control block Check if smx_MsgXchgCreate() failed.

SMXE_LQ_OVFL Indicates that the LSR queue has overflowed. It is usually due to LSRs not being allowed to run. There are several possible causes for this:

1. LSRs have been disabled by smx_LSRsOff() and not re-enabled by smx_LSRsOn().
2. An LSR is hung due to a programming error. In this case, LSRs continue to be enqueued since interrupts are enabled, but execution never returns to the LSR scheduler to run them.

3. smx_srnest is always > 1, so the LSR scheduler is never called. srnest should be 0 or a small value. If not, it has been corrupted. Watch it in the debugger.

4. Too many LSRs are being invoked due to an ISR error.

5. The processor is being overloaded by interrupts. This may be remedied by increasing SMX_CFG_LQ_SIZE in acfg.h.

SMXE_LSR_NOT_OWN_MTX Reported by smx_MutexGet() and smx_MutexRel() if called from an LSR. Mutexes are only for use by tasks.

SMXE_MMF_VIOL SecureSMX. A memory manage fault occurred.

SMXE_MTX_ALRDY_FREE Reported by smx_MutexRel() if a task tries to release a mutex that is already free. This indicates that the task has called smx_MutexRel() more than smx_MutexGet().

SMXE_MTX_NON_ONR_REL Reported by smx_MutexRel() if a task attempts to release a mutex that it does not own. Only the owner can release a mutex. A non-owner can release a mutex with smx_MutexFree() or smx_MutexClear(). But this should be done only in special situations such as recovery.

SMXE_NOT_HOLDING smx++. An smx_Msg object is not holding a message, so the Send, Put, or other operation cannot be performed.

SMXE_NULL_PTR_REF The value at address 0 has changed since initialization, which suggests a null pointer was used. This is checked in the idle task and at exit. See smxmain.c. This check is enabled by SMX_CFG_NULL_PTR_REF_CHECK in acfg.h. It should only be enabled for targets that have RAM at 0.

SMXE_OBJ_IN_USE smx++ error. Occurs when a destructor has been called and the object is still in use. See smx++ Developer's Guide.

SMXE_OBJ_NOT_CREATED smx++. Occurs when an object method is used and the object has not been created.

SMXE_OK No timeout nor error.

SMXE_OP_NOT_ALLOWED Occurs when a limited SSR is called from an LSR.

SMXE_OUT_OF_BCBS, SMXE_OUT_OF_MCBS, SMXE_OUT_OF_MUCBS, SMXE_OUT_OF_PCBS, SMXE_OUT_OF_PICBS, SMXE_OUT_OF_TCBS, SMXE_OUT_OF_TMRCBS

Out of control blocks of the type specified. This type of error occurs when a create call is unable to get a control block from its pool. Usually these errors indicate that the corresponding NUM value in acfg.h needs to be increased. For example, an SMXE_OUT_OF_TCBS error indicates that SMX_CFG_TASKS in acfg.h should be increased.

SMXE_OUT_OF_STKS The scheduler cannot get a stack from the stack pool for an unbound task. If stack scanning is not enabled, out of stacks occurs if the **freestack pool** is empty. If stack scanning is enabled, it occurs if both the freestack pool and **scanstack pool** are empty. This error is only reported the first time it occurs, to avoid cluttering the error buffer, and also since it may not be an error. This is because smx permits running lean on shared stacks.

The scheduler will run the next task in the ready queue that already has a stack (i.e. a bound task). Each time the scheduler is entered it will try to run the top unbound task again.

Eventually, the task will run when a stack becomes available. If this performance degradation is not acceptable, increase the value of SMX_CFG_STACKS in acfg.h.

SMXE_PRIV_VIOL SecureSMX. An attempt was made to call a privileged smx service from unprivileged mode, or a umode task attempted to operate on other than itself or one of its child tasks.

SMXE_Q_FIXED The scheduler detected a broken link in a ready queue level and was able to fix it.

SMXE_RQ_ERROR smx_rqtop is invalid. The scheduler attempts to fix smx_rqtop. If it fails, it reports this error, then attempts to fix the rq level. It reports SMXE_Q_FIXED if it succeeds. Otherwise the rq level is marked as empty.

SMXE_SEM_CTR_OVFL The signal counter in an event or threshold semaphore has overflowed the 0xFF limit. This error occurs on a smx_SemSignal() call. It usually indicates that the task which should be testing the semaphore is not doing so — possibly because it is being starved or due to a programming error.

SMXE_SMX_INIT_FAIL smx initialization has failed. Step through smx_Go() in your debugger to see where it fails. First, you may want to expand smx_ebi in the watch window to see the first error reported. smxAware displays the error buffer, but it may not work if not enough has been initialized before the point of failure.

SMXE_STK_OVFL Detected in the scheduler when a task is about to be stopped or suspended and stack checking is enabled for the task (task->flags.stk_chk == 1). Indicates that the task's stack pointer exceeds the stack top (task->sp < task->stp) or that the stack high water mark (task->shwm) exceeds the stack size (task->ssz). Since stack overflow is considered to be an irrecoverable error, smx_EM() calls smx_EMExitHook(), which can be programmed to perform appropriate action. If there is a stack pad, and it has not been exceeded, it is possible to continue operation.

Note: This error is logged and displayed only once per task, unless the task is restarted. It is however recorded in the global and task err and counters, every time it occurs. Another possible cause of this error is use of a foreign stack when a task switch occurs.

SMXE_TMO A timeout has occurred for the last smx service from the current task. If task->pritmo > task->pri, task->pri and task->prinorm are set to task->pritmo. This is to handle timeouts, which require a higher priority than normal such as protocol timeouts.

SMXE_TOKEN_VIOL SecureSMX. The current task does not have a proper token to create or access an object. Create and object modification operations require a HI_PRIV token.

SMXE_TOO_MANY_HEAPS The maximum number of heaps has already been created. Increase EH_NUM_HEAPS in eheap.h.

SMXE_UNKNOWN SIZE An smx peek operation cannot determine the requested size. This occurs, for example, if the size of a message made from a static block is requested, since there is no PCB.

SMXE_WAIT_NOT_ALLOWED An operation was aborted because a wait was not allowed. This occurs when an LSR makes a call which would result in waiting. LSRs must use the SMX_TMO_NOWAIT SSR timeout parameter.

SMXE_WRONG_HEAP The chunk or block pointer is outside of the range of the specified heap.

SMXE_WRONG_MODE smx_MsgXchgCreate() has an unrecognized mode.

SMXE_WRONG_POOL SecureSMX. For ARM-M v7, the block pointer or size are not multiples of the region size, or the block size < 32. For ARM-M v8, the block pointer or size are not multiples of 32, or the block size < 32.

smx ISR An ISR which interacts with smx. It must start with smx_ISR_ENTER() and it must end with smx_ISR_EXIT(). The latter calls smx_PreSched() when an smx ISR has invoked an LSR.

SMX_PRI_NOCHG No change to a task or message priority, defined in xdef.h. Used in smx_TaskBump(), smx_TaskStartNew(), and smx_Msg SSRs.

SMX_TMO_DFLT Default timeout should be a large finite value that is not expected to occur during normal operation. It should be used for all timeouts for which there is no better choice. The intent is to enable system recovery if some unexpected failure occurs.

SMX_TMO_INF Infinite timeout should be used only in cases where the default timeout would be inappropriate, such as servers that are called very seldom.

SMX_TMO_NOCHG No change to task's timeout. This can be used in any SSR with a timeout. If the task is already waiting, its timeout will not be changed. If the task is not waiting its timeout will continue to be disabled. Most oftern used with smx_TaskStop() or smx_TaskSuspend(). Has no effect on smx_ct.

SMX_TMO_NOWAIT No timeout results in a non-blocking call. LSRs must always specify this value for a timeout, since they cannot wait.

SMX_VERSION Defined in xdef.h and the processor-architecture_tool.inc file. Indicates the version of smx as 0xVVST, meaning VV.S.T. This should be used in preprocessor conditionals to handle differences in versions of smx.

SOUP Software of Unknown Pedigree. Typically applies to third party software that may not be available in source code form.

srnest **smx_srnest** is the service routine nesting level. It records the nesting level of service routines and is 1 whenever the prescheduler or scheduler is running. When an SSR starts, smx_srnest is incremented upon entry and decremented upon exit, unless it is 1. When an LSR starts, smx_srnest is incremented upon entry and decremented upon exit. When an ISR starts, smx_srnest is incremented upon entry and decremented upon exit (except for ARM-M which has RETTOBASE flag for this). Since ISRs can nest or an LSR can call an SSR or an SSR can call another SSR, smx_srnest can be larger than 1. If so, the ISR or SSR will return to the point of call, upon exit. Otherwise, an ISR or SSR will transfer control to the prescheduler and scheduler, upon exit.

SS See **system stack**.

SSP flag in heap chunk blf indicates that the last word in the chunk points to the beginning of free space at the end of the chunk. If the postchunk is freed, this space will be merged with it.

SSR **system service routine** is a function which starts with smx_SSR_ENTER() and ends with smx_SSR_EXIT(). Between these, LSRs that have been invoked by ISRs are blocked from running, so that they cannot call other SSRs. However, if an SSR calls another SSR or system function, care must be taken to not access the same smx objects. ISRs must not call SSRs and must invoke LSRs to do so.

stack Every task requires a stack when it is running or suspended. A **bound task** also requires a stack when stopped. An **unbound task** returns its stack to the stack pool when stopped. A permanent stack remains bound to a task as long as the task is not deleted. smx allows unbound tasks to be stopped while waiting for events such as signals, messages, etc. Such tasks are called one-shot tasks. Bound stacks are either pre-allocated or allocated from a heap.

stack block contains optional stack pad, stack, register save area, and optional task local storage, in that order from stack block top to bottom.

stack high water mark Actual stack usage is stored in task->shwm. The stack high-water mark indicates the maximum stack usage by the task, even if it does not have a bound stack.

stack pad is an unused space located above every task stack. Its purpose is to absorb stack overflow so the system can continue running. Its size is determined by SMX_CFG_STACK_PAD_SZ, in acfg.h. It is helpful to have a large stack pad during debugging. For release, a small stack pad, rather than no stack pad, is recommended to increase system resilience The larger the pad the greater the resilience. The stack pad is scanned ahead of the stack, so overflow into it will be detected.

stack pool is allocated from the main heap by smx_StackPoolCreate() the first time smx_TaskCreate() is called. Stacks in the stack pool are shared between unbound tasks. SMX_CFG_STACK_SIZE, in acfg.h, determines the size of **stack blocks** in the stack pool. The actual stack size is:

```
stack size = SMX_CFG_STACK_SIZE - SMX_CFG_STACK_PAD_SIZE – SMX_RSA_SIZE – SMX_CFG_TLS_SZ
```

where SMX_CFG_STACK_PAD_SIZE and SMX_CFG_TLS_SZ are defined in acfg.h, and SMX_RSA_SIZE depends upon the processor and is defined in xdef.h.

stack scan Bound stacks are filled with a known pattern when they are created. Unbound stacks are filled with the same pattern when the stack pool is created and when released by stopped tasks. Each stack is periodically scanned by the idle task, from the top of the stack pad, task->spp, to the first change of pattern. The difference between this location and task->sbp (stack bottom pointer) is compared to the stack high water mark, task->shwm and replaces it if larger. Stack scanning is a more accurate means to determine maximum stack usage versus comparing the stack pointer to the stack top when the task is suspended or stopped, because the stack pointer is not likely to be at its extreme at that time.

stack size For stacks allocated by smx_TaskCreate(), the stack size is the same as requested, except it might be slightly less due to alignment on an SB_STACK_ALIGN boundary. In this case, stack block size calculated as follows:

```
stack block size = stack size + SMX_CFG_STACK_PAD_SIZE + SMX_RSA_SIZE + SMX_CFG_TLS_SZ
```

and this is what is allocated from the heap. Stack pool and preallocated stack sizes are calculated from stack block size – see **stack pool,** above. Stacks must be large enough for the maximum nesting of functions and SSRs called by the task, but need not take into account ISR requirements, since ISRs use the system stack.

start The process of adding a task to the ready queue at the end of its priority level. The task does not actually start running until it becomes the **top task**. See smx_TaskStart() and related services.

start chunk is the first chunk in the heap. It is an 8-byte, inuse chunk with no data block. smx_heap.pi points to it.

starting bin The lowest bin that might contain a big-enough chunk. If this bin is empty, the search goes up to the first occupied higher bin.

startup code Code that runs from the processor reset vector to initialize the processor and prepare for entry into a C/C++ program. Usually it is written in assembly language. After the hardware initialization, it calls a function provided by the C compiler to clear uninitialized data, copy initialized data from ROM to RAM, call an application hook that smx uses to create smx control block pools and to initialize heaps, run C++ static initializers, and branch to main().

starvation Means that a task is not getting enough processor time to do its job. Profiling helps to identify this problem. Various strategies can be employed to correct it.

state A task can be in one of four states:

- **null** A task which has not been created is in the null state. It has no TCB and it is nonexistent for smx.
- **ready** The task is ready to run — it is in the ready queue, but not actually running.
- **run** The task is actually running. Its is still in rq.
- **wait** The task is waiting for an event to occur. It may or may not be in a queue and its timeout may or may not be set.

Only one task can be in the **run** state at a time. That task is known as the **current task** and its handle is stored in **smx_ct**. Any number of tasks may be in the other states.

statically defined means defined at compile time and assigned to memory at link time as opposed to **dynamically defined**, such as an allocation from a heap.

static block is a data block which is statically defined, e.g.:

```
u8 block[100];
```

static initializers are routines generated by a C++ compiler to initialize static (e.g. global) objects by calling their constructors. These are called during startup code after static data has been initialized, but before main() is called. Since global objects may include smx objects, all required smx control block pools are created before global object constructors are called. Also all heaps needed by the global object constructors are initialized before they are called.

stime **system time** is the 32-bit elapsed time, in seconds, from a reference time. stime is stored in smx_stime, which is initialized by sb_StimeSet(), called from ainit(). It is used by smx sleep functions and may be used to time-stamp files. The reference time is chosen by the user.

stop **task stop** ends execution of a task. When smx_ct stops itself, it is dequeued from smx_rq, put into its wait state, its stack pointer cleared, and it is enqueued on a wait queue, if expecting an event. When smx_ct stops another task with smx_TaskStop() or smx_TaskStart(), that task is dequeued from any queue it may be in, put into the wait state,

and its stack pointer is cleared. Due to clearing task->sp, the task's context is lost. If task->flags.stk_perm flag is 0, the stack is returned to the stack pool. When the specified event occurs or times out, the task is restarted from its main code beginning with a parameter equal to the return value.

stop call An SSR that causes a task to be stopped. Tasks are stopped and must be restarted, even if the expected condition is satisfied immediately. These SSRs include those that have Stop in their names and the smx_TaskStart() and smx_TaskStop() SSRs.

stuck chunk A heap chunk at the back of a large bin that is not a useful size. This can happen if cmerge is OFF, and chunk allocations from the bin are being satisfied by smaller chunks in front of it and larger sizes are being taken from the next bin.

suspend Pauses execution of a task such that it can be resumed from where it was suspended. When smx_ct suspends itself, with a suspend SSR, it is dequeued from smx_rq, its context is saved in its RSA, and it is enqueued on a wait queue, if expecting an event. When smx_ct suspends another task, using smx_TaskSuspend() or smx_TaskResume() that task is dequeued from any queue it may be in and put into the wait state. Either way, the suspended task retains its stack, and its stack pointer is saved in task->sp. When the specified event occurs or times out, the task is resumed from the point of suspension with the return value.

suspend call An SSR that causes a task to be suspended, unless the expected condition is satisfied immediately.

system function A non-SSR that performs an smx or smxBase service, for example: smx_TaskLock() and smx_PipeGet8().

system service A service provided by smx. It can be an SSR, a function, or a macro. Only SSRs are task preemption safe. All smx system services are prefixed with "smx_". smxBase services may also be referred to as system services. These are limited to functions or macros and are prefixed with "sb_". See the smxBase User's Guide.

system stack The system stack (**SS**) is used for startup, initialization (including C++ static initializers), ISRs, LSRs, the schedulers, and the error manager. SS implementation depends upon processor architecture. During initialization, it is filled with a scan pattern and it is periodically scanned by the idle task along with task stacks to determine usage. It is recommended that SS be located in on-chip SRAM for best performance. For ARM-M this stack is known as the **main stack** and it is switched to automatically due to an **exception**.

target **target system** is the hardware upon which the application software runs, as distinct from the host or development system upon which the software is developed.

task An **smx task** consists of a Task Control Block (TCB), a main function, a stack, and a timeout. A task is created by smx_TaskCreate() and can be deleted by smx_TaskDelete(). For SecureSMX a task also has a memory protection array (MPA) and it may have a token array, IRQ permission array, and portal structures.

task context consists of all register contents, TCB, task stack, and stack pointer. All of these must be preserved so the task can be resumed where it left off. Extended task context might include coprocessor registers, global variables, and other information specific to the task. These can be saved and restored with task callback function EXIT and ENTER cases.

task locking A task can be locked using smx_TaskLock() to prevent it from being preempted while in a critical section or to prevent unnecessary task switches. The lock can be removed with smx_TaskUnlock() or smx_TaskUnlockQuick().

task-safe Means that a service is safe from task and LSR preemption. SSRs achieve this because an LSR must wait for the current SSR to complete and another task cannot start until the current SSR completes. See smx User's Guide, Service Routines chapter.

task state See **state**.

task switch occurs due to preempting, stopping, or suspending the current task and starting or resuming another task. Performed by the smx task scheduler.

tc See **top chunk.**

TCB **task control block.** Each task is assigned a TCB when it is created. A TCB has many fields, which are used by smx task services and other services.

TCB pool All TCBs are in a pool, which is controlled by the **smx_tcbs** pool control block. The singly-linked list of free TCBs is pointed to by smx_tcbs.pn. The next-link pointer is in the first word of each free TCB. The last free TCB has a NULL link.

TCB_PTR task handle typedef.

temporary stack A stack given to an unbound task from the stack pool when it is dispatched. The stack is released back to the stack pool when the task stops.

thread Short for "thread of execution." Same as **task**.

thread-safe Avoidance of data races between threads.

threshold semaphore resumes the next waiting task after T signals have been received, where T is the threshold. When a task is resumed, the internal count is reduced by T. See smx User's Guide, Semaphores chapter, threshold semaphore section.

TickISRHook() callback function to hook into smx_TickISR() in smxmain.c. Can be used to piggyback ISRs on the tick interrupt for testing or to add more capability to the tick ISR without modifying it.

timeout All calls that can put tasks into the wait state permit a timeout to be specified. Timeouts ensure that tasks will not wait forever and timeouts also break task deadlocks. It is recommended that SMX_TMO_DFLT be used for all timeouts for which there is no clear choice. SMX_TMO_NOWAIT can be specified if no wait is desired. SMX_TMO_INF can be specified if no timeout is desired. SMX_TMO_NOCHG can only be used from another task with a call such as smx_TaskStop(task, SMX_TMO_NOCHG). This could be used to cause a task to stop waiting for an event, but not start until its original timeout completes.

The maximum permitted timeout is (2^31-1) ticks, which is the maximum value of etime. The resolution of task timeouts is a tick. Timeouts may be specified in milliseconds by ORing the value with SMX_FL_MSEC. The value is converted to ticks and rounded up.

timeout[] smx_timeout[] is an array of 32-bit timeouts, one per TCB. A task's timeout can be accessed via the task's index:

```
timeoutn = timeout[taskn->indx];
```

Each task timeout stores a future etime value or 0xFFFFFFFF, if it is inactive. The smallest currently active timeout is stored in smx_tmo_min, which is periodically compared to smx_etime by smx_TimeoutLSR. If smx_tmo_min is less than or equal to etime, a timeout has occurred, and smx_TimeoutLSR resumes or restarts the corresponding task. It then searches smx_timeout[] for the next smallest timeout and loads its value into smx_tmo_min and the task index into smx_tmo_indx. If it is also 0 (i.e. two tasks have timed out at once), smx_TimeoutLSR invokes itself in order to allow other LSRs to run before it runs again.

timer A timer is a system object, consisting of a timer control block (TMRCB) linked into the timer queue (smx_tq). smx supports both one-shot timers and cyclic timers. Both are created and started by smx_TimerStart(), which allows specifying a time from now to timeout and a cycle time. If the cycle time is zero, the timer is a **one-shot timer**, which is automatically deleted after it times out. Otherwise, the timer is a **cyclic timer** with the specified cycle time. Cyclic timers are requeued in smx_tq immediately so there is no cumulative timing error.

Timers are enqueued in smx_tq in order of their times, each with a calculated differential time in its TMRCB. Decrementing of the first TMRCB counter is done by smx_KeepTimeLSR. Timers have one tick resolution. When a timer times out, the specified LSR is invoked with the specified parameter. Since LSRs cannot be blocked by tasks, this provides low-jitter operation for control or sampling.

Timers can also generate pulses and be used for pulse width modulation, pulse period modulation, and frequency modulation. See smx User's Guide, Timers chapter for more information on timers.

time-slice scheduling is a task scheduling algorithm in which each time-sliced task is given a guaranteed period to run. smx provides time slicing via runtime limiting. Note that preempting tasks will not use time from a time-sliced task's period. See also preemptive scheduling and round-robin scheduling.

TLS **task local storage** is a block of memory in the task stack below the register save area, RSA. TLS should be used only with permanent stacks. It starts at task->sbp + SMX_RSA_SIZE, which can be obtained by:

```
u8* tlsp = (u8*)smx_TaskPeek(task, SMX_PK_TLSP);
```

and its size is determined by the tlssz_ssz parameter in smx_TaskCreate(), where:

```
tlssz = tlssz_ssz >> 16;
```

TLS is best accessed by defining it to be a static structure or a static array. TLS variables are not global variables and hence are safer, although smx_TaskPeek() allows obtaining the TLS pointer of another task's TLS (but not by umode tasks in SecureSMX). For SecureSMX, TLS can save an MPU slot.

TMRCB **timer control block** A timer control block is assigned to a timer and initialized when the timer is started by smx_TimerStart(). A TMRCB has forward and backward links to link

into smx_tq, owner, interval, differential count, LSR, par, and a handle pointer. These are used by timer services and smx_TimeoutLSR.

TMRCB pool All TMRCBs are in a pool, which is controlled by the smx_tmrcbs pool control block. The singly-linked list of free TMRCBs is pointed to by smx_tmrcbs.pn. The next-link pointer is in the first word of each free TMRCB. The last free TMRCB has a NULL link.

TMRCB_PTR timer handle typedef.

token **SecureSMX.** Permits access to an smx object. See SecureSMX User's Guide for more information.

top bin The last heap bin in smx_bin[]. It handles all chunk sizes from its minimum size up.

top chunk is the last chunk before the end chunk, ec, in heap hn. Initially, it and the donor chunk contain all of free heap space. Allocations which cannot be satisfied by the SBA, donor chunk, nor larger bins come from tc.

top message The first message in the message queue of an exchange.

top task The first task in the highest occupied priority level of rq. This is generally smx_ct, unless smx_ct is locked.

tq **smx_tq, timer queue** stores active timers in order of their timeouts. Each timer contains a differential count, diffcnt, such that the sum of the diffcnt' s to its position equals its timeout. It takes a little longer to enqueue a timer, but only the diffcnt of the first timer need be decremented each tick.

TRUE == 1. However, it is better to test for !0.

u8 unsigned 8-bit integer.

u16 unsigned 16-bit integer.

u32 unsigned 32-bit integer.

UBA See **upper bin array.**

umode **SecureSMX.** Unprivileged or user mode of the processor.

unbound an unbound task has no permanent stack.

unbounded priority inversion See **priority inversion**.

unlocked See **locked**

upper bin array **UBA** is that portion of bin[] array that is above the small bin array, SBA.

unrestricted macro can be invoked from any service routine or task.

utask **SecureSMX.** Unprivileged or user task, which runs in umode.

volatile registers The registers that the C/C++ compiler expects to be changed by a function call, and therefore does not depend upon them being preserved or saves them before the function call and restores them after it. See also **non-volatile registers**.

XCB **exchange control block.** Each exchange has an XCB, which contains exchange parameters. These include forward and backward links for task and message queues, mode, task queue flag, message queue flag, and name.

XCB pool All XCBs are in a pool, which is controlled by the smx_xcbs pool control block. The singly-linked list of free XCBs is pointed to by smx_xcbs.pn. The next-link pointer is in the first word of each free XCB. The last free XCB has a NULL link.

XCB_PTR exchange handle typedef.