smx®

Reference Manual

Version 4.4.0

October 2017

by Ralph Moore

Micro Digital

Micro Digital Associates, Inc.
2900 Bristol Street, #G204
Costa Mesa, CA 92626
(714) 437-7333
support@smxrtos.com
www.smxrtos.com

Revisions

date	version	comments
9/88	1.x	preliminary
12/88	1.0	first release
6/89	1.1	changes resulting from detailed cross comparison with code and resolution of inconsistencies
8/89	1.1	minor corrections
1/90	1.2	minor corrections
6/90	1.3	revision updates
2/91	2.0	major revision including addition of new calls and terms
1/92	2.1	addition of event management calls and minor corrections
4/92	2.2	changes to some macros and miscellaneous corrections
3/94	3.1	major revision and addition of new calls
5/96	3.2	simplified smx call descriptions; added protected mode and other changes; added and updated glossary entries
7/01	3.5	major revision to update to v3.5 and changes to eliminate x86 bias
5/04	3.6	update to v3.6
5/05	3.7	update to v3.7
3/07	3.7	C-only API, one-shot tasks, and minor changes
7/09	3.7	corrections and clarifications
10/10	4.0	update to v4.0
10/11	4.0	updates for new scheduler and other changes
10/12	4.1	update to v4.1 and Glossary rewrite
1/14	4.2	update to v4.2
5/14	4.2.1	update to v4.2.1
5/15	4.3	update to v4.3
2/16	4.3.1	heap
5/16	4.3.2	heap
10/17	4.4	update to v4.4

Table of Contents

smx Calls

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. This is helpful to properly use smx services.

The smx Glossary at the end of this manual defines all smx terms and symbols. It provides complete, detailed information on every aspect of smx, and should be referenced whenever the meaning of a term or symbol is in doubt.

Names in all caps are generally data types, manifest constants, macros, or enumerated constants. Names such as *atask* are handles (pointers) for objects such as tasks. Hence, to access *afield* in atask control block, we use atask->afield. It is also possible to access afield via the control block structure: tcb.afield. These two methods are equivalent and both are used. A function is identified by parentheses after the name — for example smx_TaskStart().

Format of the Calls Section

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.
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 a listed error is detected, no operation occurs except to log the error and return a failure indication, unless otherwise specified. 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 while the call is executing. Similarly, if a call is made from an LSR, that LSR is the current LSR 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 Reliability chapter of the smx User's Guide for discussion of error checking.

Notes and Restrictions

(1) Timeouts are specified in ticks but their true resolution is determined by how often smx_TimeoutLSR runs. This is configured by TIMEOUT_PERIOD in acfg.h.

(2) No task code following a stop call will not execute. Execution will continue from the start of the task's main function, including errors and timeouts. The return value is passed in as the task main parameter. Note: stop and start calls which specify the task are an exception to this.

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

(4) Bare functions can used in ISRs, but must be protected from other ISRs accessing the same object. This can be done by disabling interrupts. Bare functions do not disable interrupts.

(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) Create calls: If a name is passed, the control block's name field is set to point to it. If you manually add a name to the handle table with smx_HT_ADD(), you must remove it with smx_HT_DELETE() before deleting the object.

(7) 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). Use void if the parameter is not used — e.g. void task_main(void).

(8) LSR and task 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 LSR or task main function, as follows:

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

When the LSR or task is dispatched, the parameter is passed as data, so the value will be in a data register. If the LSR or task function were defined to take a pointer parameter, it would be expected in an address register, and the code would fail. Examples in the smx manuals are written for the typical processor and show handles as parameters (i.e. CB_PTRs).

(9) UG = smx User’s Guide.

Reference abbreviations

QS	SMX Quick Start Manual
RM	smx Reference Manual
UG	smx User’s Guide
SB	smxBase User’s Guide
SPP	smx++ Developer’s Guide

The above abbreviations may be followed by Chapter Names and section names

smx_Block and smx_BlockPool

smx_BlockGet

BCB_PTR smx_BlockGet (PCB_PTR pool, u8 **bpp, u32 clrsz)

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 to load. NULL if none.
clrsz Number of bytes to clear from the start of block.

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

Errors
SMXE_INV_PCB
SMXE_OUT_OF_BCBS

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 is intended to be used to load data into the data block. The current task or LSR becomes the smx block owner. Returns the block handle. If pool is invalid or if out of BCBs, aborts, returns NULL, and bpp is not changed.

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 the specified pool, loads data into it, and returns the block handle.

smx_BlockMake

BCB_PTR smx_BlockMake (PCB_PTR pool, u8 *bp)

Type SSR

Summary Makes a block from a bare block using its pointer.

Compl smx_BlockUnmake()

Parameters

pool	Base pool of the block. NULL if none
bp	Block pointer.

Returns

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

Errors SMXE_OUT_OF_BCBS
SMXE_INV_PARM

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 is stored in the BCB (NULL if no pool). If pool != NULL, bp is range-checked; if pool == NULL, bp is checked for non-zero. If either test fails operation is aborted and NULL is returned.

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

in_pipe = smx_PipeCreate(&pp, WIDTH, LENGTH, "in_pipe");

void inISR(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(inLSR,(u32)bp)
         break;
      default:
         *dp++ = ch;
   }
}

void inLSR(u32 bp);
{
   BCB_PTR blk;

   blk = smx_BlockMake(&in_pool, (u8*)bp);
   if (!smx_PipePutWait(in_pipe, &blk, NO_WAIT))
   smx_BlockRel(blk, 0);
} /* eb8 */
```

inISR() runs whenever an 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, inLSR() is invoked. inLSR() 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_PARM par)

Type SSR

Summary Returns the current value for the argument specified.

Parameters

blk	Block to peek at.
par	What to peek at.

Returns

value	Value of par.
0	Value, unless error.

Errors SMXE_INV_BCB
SMXE_INV_PARM

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)

Type SSR

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

Compl smx_BlockGet()

Parameters blk Block to release.

Returns

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

Errors SMXE_INV_BCB
SB_INV_POOL
SB_INV_BP

Descr Releases a block obtained by smx_BlockGet() or made by smx_BlockMake. Aborts immediately if blk is invalid or it has already been released. Otherwise, releases the data block back to its base pool, if it is a base block. In this case, the base pool must be valid, and the block pointer must point within it, otherwise the release is aborted. Clears clrsz bytes up to the end of the block. Also releases the BCB back to its pool. Returns TRUE if successful. Fails and returns FALSE, otherwise.

Notes

1. This SSR can be used to release an smx block, which was made from a bare block.
2. This SSR will not produce reliable results if blk has already been released or unmade because the BCB may have already been assigned to another smx block. To guard against this happening, clear its handle (blk) immediately after a BCB has been returned to its pool.
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 Number of blocks released.

Errors SMXE_INV_TCB

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

Example

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

Unlike smx_TaskDelete(&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 may be necessary because atask may own one or more blocks when another task stops it.

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()

Parameters

	pool	Place to put pool handle if it is not NULL.
	blk	Block to unmake.

Returns

	>0	Block unmade.
	NULL	Invalid BCB or blk has already been unmade or released.

Errors SMXE_INV_BCB

Descr Unmakes an smx block made by smx_BlockMake() or a block obtained by smx_BlockGet() by converting it into a bare block by releasing its BCB. Returns NULL if the BCB is invalid or the block has already been unmade or released. Otherwise, returns the address of the data block and loads its pool handle into the user-supplied pool location, unless NULL. (For a base block, the code receiving the block must know its pool handle.)

Notes

1. Interrupt safe with respect to sb_BlockGet() and sb_BlockRel() operating on the same block pool.
2. Following unmake, use sb_BlockRel() to release the base or smx data block back to its pool, when no longer needed.

Example

```
PICB_PTR    out_pipe;
u8 *        pkt_ptr;
PCB_PTR     pkt_pool;
u32         pkt_sz;
```

```
smx_LSRInvoke(outLSR, (u32)out_pipe);

void outLSR(u32 pipe);
{
    BCB_PTR blk;
  if (smx_PipeGet((PICB_PTR)pipe, (u8*)&blk))
    {
        pkt_sz = smx_BlockPeek(blk, SMX_PK_SIZE);
        pkt_pool = (PCB_PTR)smx_BlockPeek(blk, SMX_PK_POOL);
        pkt_ptr = smx_BlockUnmake(&pkt_pool, blk);
        StartISR(outISR);
    }
}
void outISR(void);
{
        if (pkt_sz > 0)
        {
        UART_Out(*pkt_ptr);
        if (pkt_sz--)
            pkt_ptr++;
    }
    else
    {
        UART_Stop();
        sb_BlockRel(pkt_pool, pkt_ptr, 0);
    }
} /* eb9 */
```

This example is the opposite of that shown for smx_BlockMake(). It is assumed that a task invokes outLSR() whenever it puts a block handle into out_pipe. outLSR() gets the next block handle from out_pipe, determines its size, then unmakes it into a bare block at pkt_ptr. Note that the pool handle from the BCB is loaded into pkt_pool. outLSR() then starts outISR(). The UART interrupts each time it needs another character and outISR provides the character until all characters have been sent. outISR then stops the UART and releases the bare block back to pkt_pool. Note that 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. Regardless of how blk was formed, it is released to where it belongs.

Notes:

1. Using outLSR() is not essential — its functions could be performed by a task.
2. The data block is not cleared, when it is released, in order to minimize ISR execution time.
3. If, for some reason, there is no block handle in out_pipe, nothing happens.

smx_BlockPoolCreate

PCB_PTR smx_BlockPoolCreate (u8 *p, u8 num, u16 size, const char *name)

Type	SSR		
Summary	Creates an smx block pool of num size blocks at bp.		
Compl	smx_BlockPoolDelete()		
Parameters	p	Pointer to memory for pool.	
	num	Number of blocks.	
	size	Block size.	
	name	Name to give block pool, NULL for none.	
Returns	pool	Pool handle.	
	NULL	Insufficient resources or error.	
Errors	SMXE_INV_PAR	p is not > 0 and a multiple of SB_DATA_ALIGN, num is not > 0, size is not > 0 and a multiple of SB_DATA_ALIGN.	
	SMXE_INSUFF_DAR	Unable to create PCB, BCB, or MCB pool.	
	SMXE_OUT_OF_PCBS		

Descr Tests for valid parameters and returns NULL, if not. Creates PCB, BCB, and MCB pools, if not already created, and gets PCB for pool. If fail any of these, reports error and returns NULL. Calls sb_BlockPoolCreate() to creates block pool of num*size blocks at p. Loads name. If successful, initializes PCB and returns block pool handle. If not, returns PCB to its pool and returns NULL.

SB_DATA_ALIGN is defined in the processor architecture header file (e.g. barm.h). It normally is 4 bytes for 32-bit processors. For processors, which cannot write a 32-bit value to a byte boundary, such as ARM, attempting to write a 32-bit value to the start of an unaligned block will corrupt memory below the block, and writes within the block may not go where expected. Even if a processor can write a word on a byte boundary, performance is greatly improved by aligning all blocks on 4-byte boundaries.

Note The block from which the block pool is created can be allocated in any manner and can be anywhere in memory, even in ROM. It is assumed to be large enough for the block pool. The block pool does not require any extra bytes — no padding is introduced.

Example

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

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

if ((poolA = smx_BlockPoolCreate(p, NUM, SIZE, "poolA") != NULL)
      /* proceed */
```

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_BlockPoolCreateDAR

PCB_PTR smx_BlockPoolCreateDAR (SB_DCB_PTR dar, u8 num, u16 size, u16 align, const char *name)

Type SSR

Summary Allocates block for pool from dar and creates a block pool of num size blocks.

Parameters

dar	Handle of DAR Control Block (DCB)
num	Number of blocks.
size	Block size.
align	Alignment in bytes (e.g. 4 or SB_CACHE_LINE)
name	Name to give block pool, NULL for none.

Returns

pool	Handle of pool created.
NULL	Insufficient resources or error.

Errors

SMXE_INSUFF_DAR	Unable to create pool or PCB, BCB, or MCB pool.
SMXE_INV_PAR	num is not > 0, size is not > 0 and a multiple of SB_DATA_ALIGN.
SMXE_OUT_OF_PCBS	

Descr Performs the same function as smx_BlockPoolCreate() except that it allocates space for the pool from a DAR. If it fails, the pool space is returned to the DAR. The advantage of using this SSR is that it is more automatic than smx_BlockPoolCreate(). There is no need to define a static block or to get a block from the heap and to make sure that the block is the right size and properly aligned for the pool. By properly defining DARs, it is possible to put the pool in the best memory for its usage (e.g. fast memory if the pool is actively used).

Example

```
#define NUM 100;
#define SIZE 20;
SB_DCB_PTR sb_adar;
PCB_PTR poolA;

if ((poolA = smx_BlockPoolCreateDAR(sb_adar, NUM, SIZE, SB_DATA_ALIGN, "poolA") != NULL)
    /* proceed */
```

This does the same thing as the smx_BlockPoolCreate() example, except that the pool is taken from ADAR.

smx_BlockPoolDelete

u8 *smx_BlockPoolDelete (PCB_PTR *pool)

Type SSR

Summary Deletes an smx block pool.

Compl smx_BlockPoolCreate() and smx_BlockPoolCreateDAR()

Parameters pool Pointer to pool handle.

Returns
>0 Pool deleted.
NULL Error.

Errors
SMXE_BLK_IN_USE One or more blocks are in use.
SMXE_INV_PCB

Descr Deletes a block pool created by smx_BlockPoolCreate() or smx_BlockPoolCreateDAR(). Aborts if pool is invalid or one or more blocks are still in use; reports error and returns NULL. Otherwise clears and releases PCB, clears pool handle, and returns a pointer to the start of the released pool block.

Note User is responsible for dealing with the pool block. If it came from the heap it can be freed, but otherwise it must be reused.

Example

```
void *   bp;
PCB_PTR  poolA;

bp = (void*)smx_BlockPoolDelete(&poolA);
smx_HeapFree(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 error.

smx_BlockPoolPeek

u32 smx_BlockPoolPeek (PCB_PTR pool, SMX_PK_PARM par)

Type	SSR	
Summary	Returns the current value for the parameter specified.	
Parameters	pool	Block pool to peek at.
	par	Parameter to return.
Returns	value	Value of par.
	0	Value, unless error.
Errors	SMXE_INV_PCB	Invalid block pool handle.
	SMXE_INV_PARM	Invalid argument.

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_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_FREE);
    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_CLib

smx_CLibEnter

BOOLEAN smx_ClibEnter (void)
BOOLEAN smx_ClibEnterNoWait (void)

Type Macro that maps to SSR

Summary Tests the in_clib semaphore using smx_SemTest() with timeout SMX_TMO_INF or SMX_TMO_NOWAIT, respectively. One of these should be used immediately before calls to functions in the C run-time library that are not known to be reentrant. Some C run-time libraries are reentrant (because they are protected by a semaphore), so for them, the macro can be defined to do nothing.

in_clib is defined and created in the Protosystem. It is not an smx global since it is not needed if your C run-time library is reentrant. Also, you may wish to define several semaphores, instead of just this one, to protect different groups of related functions in the C library.

Parameters none

Returns

TRUE	Ok.
FALSE	Error or timeout.

Example

```
smx_CLibEnter()
_ltoa(ctr, &buffer, 10);
smx_CLibExit()
```

smx_CLibExit

BOOLEAN smx_ClibExit (void)

Type Macro that maps to SSR

Summary Signals the in_clib semaphore using smx_SemSignal(). This should be used immediately after calls to functions in the C run-time library that are not known to be reentrant. See smx_CLibEnter().

Parameters none

Returns	TRUE	Signal sent.
	FALSE	Error.

Example See smx_CLibEnter()

smx_Conv

smx_ConvLinearToPointer

u8 ***smx_ConvLinearToPointer (u32 linear_addr)

Type Unrestricted macro

Summary Converts a linear address to a pointer. In flat memory architectures, a linear address and pointer are equal. This may not be true if the memory management unit (MMU) is used.

Parameters linear_addr Linear address to convert.

Returns pointer

Example

```
u8 *ptr = smx_ConvLinearToPointer(0x50000);
```

Returns 0x50000.

smx_ConvPointerToLinear

u32 smx_ConvPointerToLinear (void *ptr)

Type Unrestricted macro

Summary Converts a pointer to a linear address. In flat memory architectures, a linear address and pointer are equal. This may not be true if the memory management unit (MMU) is used.

Parameters ptr Pointer to convert.

Returns linear address

Example

```
u32 linaddr = smx_ConvPointerToLinear(0x50000000);
```

Returns 0x50000000.

smx_ConvMsecToTicks

int smx_ConvMsecToTicks (int msec) (rounded up)
int smx_ConvMsecToTicksRound (int msec)(rounded to nearest)

Type Unrestricted macros

Summary Converts milliseconds into ticks, rounded up to the next tick or rounded to the nearest tick, respectively. The precision of the conversion depends on the tick rate. Low tick rates result in poor precision. For example, at 10 ticks/sec, 1 tick is 1000/10 == 100 msec. At 250 ticks/sec, 1 tick is 4 msec. A tick rate of 1000 gives exact conversion.

Parameters msec Time in milliseconds to convert.

Returns time in ticks

Example

```
u32 nticks = smx_ConvMsecToTicks(25);
```

Converts 25 milliseconds into the number of ticks, at the tick rate specified by smx_cf.sec, rounded up to the nearest integer. For a tick rate of 100Hz, nticks = 3.

smx_ConvSecToTicks

int smx_ConvSecToTicks (int sec)

Type Unrestricted macro

Summary Converts seconds into ticks. This macro is intended to convert small differential times, not smx_stime, which could cause overflow.

Parameters sec Time in sec to convert.

Returns time in ticks

smx_ConvTicksToMsec

int smx_ConvTicksToMsec (int ticks)	rounded up
int smx_ConvTicksToMsecRound (int ticks)	rounded to nearest value

Type unrestricted macro

Summary 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 differential time, not smx_etime, which could cause overflow.

Parameters ticks Time in ticks to convert.

Returns time in milliseconds

smx_ConvTicksToSec

int smx_ConvTicksToSec (int ticks)	rounded up
int smx_ConvTicksToSecRound (int ticks)	rounded to nearest value

Type Unrestricted macro

Summary Converts ticks into seconds, rounded up to the next second or rounded to the nearest second, respectively.

Parameters ticks Time in ticks to convert.

Returns time in sec

smx_Delay

smx_DelayMsec

BOOLEAN smx_DelayMsec (int msec)

Type Macro that maps to SSR

Summary Delays for at least the specified number of milliseconds, as close as possible, to the precision of the tick rate. The precision of the delay is 1 tick, which may be many milliseconds, depending on the tick rate. Uses smx_ConvMsecToTicks() to convert to ticks, adds 1, then calls smx_EventQueueCount() to do the delay. May be used only from tasks, not ISRs since it calls an SSR, and not from LSRs since it waits. The main purpose of this routine is for use in drivers, for example, where the specification requires a delay of some number of milliseconds. It is understood that the delay must be at least that number of milliseconds.

Parameters msec Time to delay, in milliseconds.

Returns TRUE Delay completed.
FALSE Error.

Example

```
smx_DelayMsec(5);
```

This delays 5 milliseconds as close as possible for the tick rate. For a 100 Hz tick rate, this would delay between 10 and 20 milliseconds.

smx_DelaySec

BOOLEAN smx_DelaySec (int sec)

Type Macro that maps to SSR

Summary Delays for the specified number of seconds, using smx_EventQueueCount(). Adds 1 tick so the delay is at least as long as intended, since the next tick may be just about to occur. It is unlikely this delay needs to be so precise, but adding 1 tick is done for consistency with the other smx_Delay macros. This macro is provided for completeness and to make user code more readable than calling smx_EventQueueCount().

Parameters sec Time to delay, in sec.

Returns TRUE Delay completed.
FALSE Error.

Example

```
smx_DelaySec(3);  /* wait 3 seconds */
```

smx_DelayTicks

BOOLEAN smx_DelayTicks (int ticks)

Type Macro that maps to SSR

Summary Delays for the specified number of ticks, using smx_EventQueueCount(). Adds 1 tick so the delay is at least as long as intended, since the next tick may be just about to occur. There is probably not much need for this macro, but it is provided for completeness and to make user code more readable than calling smx_EventQueueCount().

Parameters msec Time to delay, in ticks.

Returns

TRUE	Delay completed.
FALSE	Error.

Example

```
smx_DelayTicks(100);  /* wait 100 ticks */
```

smx_ERROR

smx_ERROR

void smx_ERROR (SMX_ERRNO errnum, VOID_PTR handle)

Type Macro calling a bare function

Summary This is the smx error service macro. It switches to System Stack, then calls smx_EB(), which is the smx error manager. smx_EB() saves errnum in smx_errno and in the current task's err field; it increments smx_errctr and smx_errctrs[errnum]; it makes entries in EB and EVB; and it enqueues an error message for later display. Certain errors may also result in the current task being restarted or the system being rebooted. These are determined by the value of errnum. See UG Error Manager for more discussion.

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_EB(). 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, will show full error messages, else just error numbers. Normally called when the user presses ^E at the terminal. Should be called only from a low-priority task because it polls the UART to send characters.

smx_EVB

smx_EVBInit

void smx_EVBInit (u32 *flags*)

Type Bare function

Summary Creates and initializes the Event Buffer.

Parameters flags Flags to 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_SSR1-8 SSR groups 1-8.
SMX_EVB_EN_SSRS All SSR groups.
SMX_EVB_EN_ALL.

Returns none

Descr Initializes the event buffer and specifies which types of events to log. See the EVB_EN constants in xevb.h. The EVB_EN flags can also be changed via smxAware. See UG Event Logging for more discussion.

Example

```
void smx_Go(void)
{
    ...
    smx_EVBInit(EVB_EN_ALL);  /* enable logging of all events */
    ...
}
```

smx_EVB_LOG

void smx_EVB_LOG_ISR (u32 handle)
void smx_EVB_LOG_ISR_RET (u32 handle)
void smx_EVB_LOG_LSR (u32 handle)
void smx_EVB_LOG_LSR_RET (u32 handle)
void smx_EVB_LOG_SSR0 (u32 id) to smx_EVB_LOG_SSR6 (u32 id, u32 par1, …, par6)
void smx_EVB_LOG_USER0 (u32 handle) to smx_EVB_LOG_USER6 (u32 handle, u32 par1, …, par6)

Type Bare macros and functions

Summary Add events to the Event Buffer.

Parameters

	handle	Task handle or LSR/ISR pseudo handle.
	id	SSR ID
	par	SSR parameter or User value.

Returns None

Descr These macros add events to the Event Buffer. Most events are logged automatically by smx, but it is necessary to bracket ISRs and LSRs with the macros shown above in order to log their entry and exit. The user event can be used anywhere in the code to serve as a timestamp and to show the values of up to six variables.

The ISR and LSR macros are written as macros, for speed; the others call functions, to minimize code space. Functions that can be called from assembly are provided for use from assembly ISRs and LSRs. 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, not in saving/restoring registers and in smx_ISR_ENTER() and smx_ISR_EXIT().

Example

```
void appl_init(void)
{
    VOID_PTR  lsr1_h = create_pseudo_handle();  /* LSRs do not have handles */
    smx_HT_ADD(lsr1_h, "lsr1");
}

void lsr1(u32 par)
{
    smx_EVB_LOG_LSR(lsr1);
    /* lsr1 code */
    smx_EVB_LOG_LSR_RET(lsr1);
}
```

This example shows how to log an LSR into the Event Buffer. A pseudo handle is created for it because LSRs do not have handles. Logging an ISR is similar.

smx_EventFlags and smx_EventGroup

smx_EventFlagsPulse

BOOLEAN smx_EventFlagsPulse (EGCB_PTR eg, u16 pulse_mask)

Type SSR

Summary Pulses event flags on and off that were 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 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 AND   SMX_EF_AND
#define F2    0x2
#define F1    0x1
EGCB  eg;

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

Resumes tasks already waiting for F2&F1. Neither flag is left set if it was not already set.

smx_EventFlagsSet

BOOLEAN smx_EventFlagsSet (EGCB_PTR eg, u16 set_mask, u16 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 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 event group, 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 (including ANDOR and AND) and post_clear_mask are obtained from its TCB. The test mask is compared to eg->flags and if there is a match, the task is resumed (i.e. moved to rq). 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. 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. The result of the AND is the reset mask for the task.

If there are multiple tasks waiting, the above procedure is repeated for each one. 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 filtering by corresponding post_clear_masks, are reset. See smx_EventFlagsTest() and UG Event Groups for more discussion and examples.

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, u16 post_clear_mask, u32 timeout)

Type SSR

Summary Tests for a match between eg->flags and test_mask. If found, ct is continued, returns the flags causing the match, and clears those selected by the post_clear_mask. Suspends ct if no match is found and timeout > 0.

Compl smx_EventFlagsSet() and smx_EventFlagsPulse().

Parameters

eg	Event group.
test_mask	Flags to test.
post_clear_mask	Flags to reset of those causing a match.
timeout	Timeout in ticks.

Returns

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

Errors

SMXE_INV_EGCB	Invalid event group handle.
SMXE_INV_PARM	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. If test_mask bit 17 is 1 (0x20000), tests for the AND/OR of flags (AND/OR overrides AND). Both bits zero specifies to test for the OR of flags. Bits 15 - 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 LSB, as possible — e.g. ABC in bits 2, 1, and 0.

If there is no match, and timeout > 0, saves test_mask and post_clear_mask in ct->sv, sets ct->flags.ef_andor, if ANDOR, sets ct->flags.ef_and, if AND, enqueues task in eg wait queue, in FIFO order, loads ct timeout, and suspends ct. Suspend is not allowed for an LSR. Fails and returns 0 if no match and no timeout or LSR.

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

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. What it can do is to schedule itself, via a timer, to test again in the future.

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
EGCB_PTR   modem_eg;

void transmitMain(void)
{
    while (smx_EventFlagsTest(modem_eg, AND+TXRDY+DSR+CTS, TXRDY+DSR+CTS, tmo))
    {
        /* send next message */
    }
}
```

The transmit task waits for the modem_eg flags: TXRDY, DSR, and CTS to all be TRUE. The flags are auto-reset, then the task sends the next message and waits upon the flags, again. This example could be improved by adding:

```
#define MRDY0x8/* message ready */

while (smx_EventFlagsTest(modem_eg, AND+TXRDY+DSR+CTS+MRDY,
                                   TXRDY+DSR+CTS+MRDY, INF))
{
    /* send next message */
}
```

MRDY is set by the task, which is preparing messages to send:

```
smx_EventFlagsSet(modem_eg, MRDY, 0);
```

and the other flags are set by a UART LSR.

smx_EventFlagsTestStop

void   smx_EventFlagsTestStop (EGCB_PTR eg, u32 test_mask, u16 post_clear_mask, u32 timeout)

Type Limited SSR — tasks only.

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

Compl smx_EventFlagsSet() and smx_EventFlagsPulse().

Parameters

eg	Event group.
mask	Flags to test.

	post_clear_mask	Flags to reset of those causing match.
	timeout	Timeout in ticks.

Errors	SMXE_OP_NOT_ALLOWED	Called from LSR
	SMXE_INV_EG	Invalid event group handle.
	SMXE_INV_PARM	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 LSR.
(2) smx_lockctr is cleared if called from a task.

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 > 0)
        /* send message */
    smx_EventFlagsTestStop(modem, AND+TXRDY+DSR+CTS, TXRDY+DSR+CTS, tmo);
}
```

This task would initially be started as follows:

```
    TCB_PTR transmit;
    smx_TaskStart(transmit, 0);
```

The first time transmit runs, it will test the modem flags and stop. If there is a match, it will immediately restart and the matching flags will be passed into transmitMain() as par. Otherwise, it will wait for a match. In this example, it is assumed that if there is an error or timeout, the correct solution is to try again.

smx_EventGroupClear

BOOLEAN smx_EventGroupClear (EGCB_PTR eg, u16 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, F1);
```

Clears eg task wait list and leaves eg->flags = F1.

smx_EventGroupCreate

EGCB_PTR smx_EventGroupCreate (u32 init_mask, const char *name)

Type SSR

Summary Creates an event group with 16 flags.

Compl smx_EventGroupDelete()

Parameters

init_mask	Initial values of flags.
name	Name to give event group (NULL for none).

Returns

handle	Event group created.
NULL	Event group not created.

Errors SMXE_OUT_OF_QCBS

Descr Allocates an event group control block from the QCB pool and initializes it. If allocation fails because no block is available, returns NULL. Loads init_mask into EGCB flags field and name into EGCB name field. Rest of fields are cleared.

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 *eg)

Type SSR

Summary Deletes an event group.

Compl smx_EventGroupCreate()

Parameters *eg Address of the handle of the event group to delete.

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 QCB pool, removes the event group name from HT, and clears the eg handle (i.e. *eg == NULL) so eg cannot be used again.

Example

```
EGCB_PTR eg;

smx_EventGroupDelete(&eg);
```

smx_EventGroupPeek

u32 smx_EventGroupPeek (EGCB_PTR eg, SMX_PK_PARM par)

Type SSR

Summary Returns the current value for the parameter specified.

Parameters

eg	Event group to peek.
par	Parameter to return.

Returns

value	Value for par.
0	Value, unless error..

Errors

SMXE_INV_EGCB
SMXE_INV_PARM

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 = NULL;
```

.

smx_EventQueue

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

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)

Type limited SSR — tasks only

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

Compl smx_EventQueueSignal()

Parameters	eq	Event queue to wait at.
	count	Number of events to wait for.
	timeout	Timeout in ticks.

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

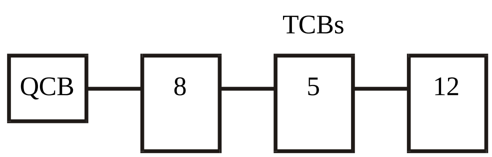
Errors

SMXE_OP_NOT_ALLOWED
SMXE_INV_EQCB
SMXE_BROKEN_Q

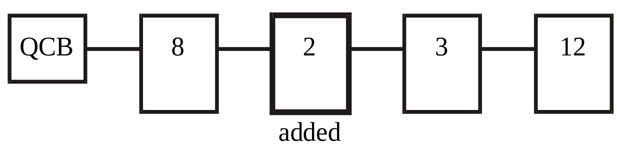
Descr

If count is 0 or timeout is 0, continues the current task and returns FALSE. If both are 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:



If smx_EventQueueCount() is used to count ticks, it can produce delays which are accurate to one tick. Overhead is proportional to the length of the queue and is only incurred on each smx_EventQueueCount() call. The overhead per signal to eq is not affected by the length of the queue, unless many tasks are resumed by one signal. See smx_DelayMsec() macro.

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
XCB_PTR  out_port1, in_port1, pool;
EQCB_PTR  msg_rec;

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

void statusMain(void)
{
    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_MsgSendPR(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.

Example 2

```
int pulses = 5;

void alarmMain(void)
{
    if (pulses--)
    {
        turn_on_alarm();
        smx_EventQueueCount(smx_TicksEQ, 100, SMX_TMO_INF);
        turn_off_alarm();
        smx_EventQueueCount(smx_TicksEQ, 100, SMX_TMO_INF);
    }
}
```

In this example, an alarm is on 100 ticks, then off 100 ticks. This is repeated 5 times. smx_TicksEQ is a standard event queue which is signaled every tick. When done, the alarm task autostops.

smx_EventQueueCountStop

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

Type limited SSR — tasks only

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

Compl smx_EventQueueSignal()

Parameters

count	Number of events to wait for.
eq	Event queue to wait at.
timeout	Timeout in ticks.

Errors
SMXE_OP_NOT_ALLOWED
SMXE_INV_EQCB
SMXE_BROKEN_Q

Descr See smx_EventQueue() for operational description. 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) smx_lockctr is cleared if called from a task.

TaskMain void task_main(BOOLEAN par)

par

TRUE	Count completed.
FALSE	Zero count or timeout, or timed out.

Example 1

```
XCB_PTR  out_port1, in_port1, pool;
EQCB_PTR  msg_rec;
BOOLEAN  start;

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

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

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

This example is equivalent to that shown for smx_EventQueueCount(). The start flag is necessary to get the status task to wait on msg_rec the first time. See UG One Shot Tasks for more discussion of one-shot tasks.

Example 2

```
BOOLEAN on = FALSE;
int pulses = 5;

void alarmMain(void)
{
    if (pulses--)
    {
        if (!on)
        {
            turn_on_alarm();
            on = TRUE;
            smx_EventQueueCountStop(smx_TicksEQ, 100, SMX_TMO_INF);
        }
        else
        {
            turn_off_alarm();
            on = FALSE;
            smx_EventQueueCountStop(smx_TicksEQ, 100, SMX_TMO_INF);
        }
    }
}
```

This example is equivalent to that shown for smx_EventQueueCount(). Note that ct stops, then restarts after 100 ticks. The global variables on and pulses control operation. This is a good application of a one-shot task — it uses a stack briefly then releases its stack for 100 ticks.

smx_EventQueueCreate

EQCB_PTR smx_EventQueueCreate (const char *name)

Type	SSR	
Summary	Creates an event queue.	
Compl	smx_EventQueueDelete()	
Parameters	name	Name to give event queue or NULL for none.
Returns	handle	Event queue created.
	NULL	Event queue not created due to Error.
Errors	SMXE_OUT_OF_QCBS	

Descr Gets a queue control block from the QCB 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. It would be signaled from an LSR or task with:

```
smx_EventQueueSignal(SignalsEQ);
```

smx_EventQueueDelete

BOOLEAN smx_EventQueueDelete (EQCB_PTR *eq)

Type SSR

Summary Deletes an event queue.

Compl smx_EventQueueCreate()

Parameters eq Event queue to delete.

Returns

TRUE	Event queue deleted or eq already NULL.
FALSE	Error.

Errors SMXE_INV_EQCB

Descr Deletes an event queue created by smx_EventQueueCreate(). First resumes all waiting tasks, with FALSE return values and clearing their timeouts. Then clears the EQCB, releases it to the QCB pool, and clears eq.

Example

```
EQCB_PTR smx_TicksEQ;

smx_EventQueueDelete(&smx_TicksEQ);
```

smx_EventQueueSignal

BOOLEAN smx_EventQueueSignal (EQCB_PTR eq)

Type SSR

Summary Signals an event queue. The top task waiting there may be resumed.

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_QCB
SMXE_WRONG_TYPE_QUEUE

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

Example 1

```
EQCB_PTR  revs;
TCB_PTR  wheel;

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

void revLSR(u32 par)
{
    smx_EventQueueSignal(revs);
}

void wheel_main(void)
{
    while (smx_EventQueueCount(revs, N, INF)
    {
        /* perform N revs operation */
    }
}
```

Each time a wheel turns, it causes a pulse, which triggers an interrupt and revISR runs. It invokes the revLSR, which signals the revs event queue. The wheel task runs every N revolutions.

smx_Go

smx_Go

void   smx_Go (void)

Type function

Summary Initializes smx from information in the *configuration table, smx_cf.*

Parameters none

Returns none

Errors SMXE_INSUFF_DAR
SMXE_SMX_INIT_FAIL

Descr smx_Go() allocates space for smx structures and creates task timeout array, LSR queue, stack pool, ready queue, smx_TicksEQ, smx_ts semaphore, and smx_Idle. It then starts smx_Idle with ainit() as its main function and begins operation in the task environment. 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 smx_cf, which should be initialized in main.c from constants in acfg.h. These control the amounts of memory used by smx control blocks and other smx structures, as well as the tick rate, stack parameters, and other smx features.

Example

```
void main(void)
{
    sb_IRQsMask();
    smx_Go();
}
```

It is important to mask interrupts until ainit() begins running.

smx_Heap

The following heap services are provided by the eheap embedded system heap. They meet the ANSI C/C++ Standard for malloc(), free(), realloc(), and calloc() and offer additional services, which are described in this section. See the Heap and Heap Management chapters in the smx User's Guide for more information concerning eheap and how to best use these services.

smx_HeapBinPeek

u32 smx_HeapBinPeek (u32 binno, SMX_PK_PARM par)

Type SSR

Summary Returns information concerning the selected heap bin.

Parameters

binno	Bin number.
par	What to peek at.

Returns

value	Value of par.
0	Value, unless error.

Errors SMXE_INV_PARM Invalid parameter.

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

SMX_PK_COUNT	Number of chunks in bin.
SMX_PK_FIRST	Address of first chunk in bin, NULL if empty.
SMX_PK_LAST	Address of last chunk in bin, NULL if empty.
SMX_PK_SIZE	Minimum chunk size for bin.
SMX_PK_SPACE	Free space in bin.

HeapBinPeek() reports SMXE_INV_PARM and returns 0, if par is not one of the above or if binno is not in the range 0 to smx_top_bin. It also returns 0 if the bin is empty, except for SMX_PK_SIZE. This service is an SSR. Using it is recommended over directly reading bin parameters, which can result in incorrect readings due to preemption by another task.

Example

```
CCB_PTR cp;

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

This example gets a pointer to the first chunk in bin 14.

smx_HeapBinScan

BOOLEAN smx_HeapBinScan (u32 binno, u32 fnum, u32 bnum)

Type	SSR	
Summary	Scans forward through free chunk list of binno for broken links and fixes what it can. Can scan 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.
Returns	TRUE	Done or unfixable error encountered.
	FALSE	Continue scanning.
Errors	SMXE_HEAP_BRKN	Cannot fix heap.
	SMXE_HEAP_FIXED	A heap fix was made.
	SMXE_INV_PARM	Invalid parameter.

Descr smx_HeapBinScan() scans the specified bin free chunk list and fixes broken links that it finds or reports SMXE_HEAP_BRKN if a link is unfixable. Normally it is called once per pass of the idle task and scans fnum chunks forward, per run. It is an SSR, so it cannot be interrupted by another heap service, during a run. Scans are broken into runs, to permit higher priority tasks to run, without missing their deadlines. If binno is out of range, or if either fnum or bnum is 0, SMXE_INV_PARM is reported and TRUE is returned.

A global pointer, smx_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. smx_bsp is set to NULL by smx_HeapInit() and also 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.

If the bin is empty, TRUE is returned immediately. If broken, the bin's free back link, fbl, is fixed first. If the bin ffl is out of heap range, it and fbl are set to NULL and the bin's bmap bit is also cleared. This makes the bin empty. Then SMXE_HEAP_BRKN is reported and TRUE is returned. In this case, the chunks that were in the bin are no longer available for allocation, however the heap can continue operating. If cmerge is ON, these chunks may eventually be merged with other chunks, as they are freed, and thus their free memory becomes available, again. Therefore, it may not be necessary to take further action, in this case.

If the bin has only one-chunk, TRUE is returned after fixing broken links and binx8 in the chunk, if necessary.

If the bin has more than one chunk, cp is advanced one chunk at a time until fnum chunks have been checked or the end of the bin free list has been reached. The free forward link of each chunk is heap-range checked before use. If it fails, smx_heap.mode.bs_fwd is turned OFF, the smx_bfp (bin fix pointer) is set to the end of the binno free list, and FALSE is returned. Thereafter, when smx_HeapBinScan() is called, it will scan backward bnum chunks, at a time, fixing broken ffl's, as it goes, until it reaches smx_bsp. Then smx_heap.mode.bs_fwd is turned back ON and FALSE is returned. Thereafter, when smx_HeapBinScan() is called, forward scan will resume from smx_bsp. Normally, only the one broken ffl will need to be fixed – i.e. the one at smx_bsp. If no further broken links are found, forward scan will continue, fnum chunks per run, until the end of the heap is reached. Then the scan stops and TRUE is returned.

If a free() preempts between runs it may add a chunk to the start or the end of the bin free list, but this does not affect either smx_bsp or smx_bfp, so nothing is done. If a malloc() preempts between runs, it may take the chunk pointed to by either scan pointer. In this case, the scan is aborted and a new forward scan is started from the beginning of the bin free chunk list.

If the backward scan finds a broken back link before it reaches smx_bsp, then it is not possible to fix the broken forward link at smx_bsp. So, instead, the gap is bridged from smx_bsp to smx_bfp and SMXE_HEAP_BRKN is reported. The bridge allows the scan to finish and the heap can continue operating. This is like the broken bin ffl, above, but only part of the bin has been lost. See UG Heap Management chapter for more information.

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, SMXE_HEAP_FIXED is reported, and the scan continues.

Example

```
void IdleMain(void)
{
    static u32 i = 0;
    ..
    if (smx_HeapBinScan(i, 2, 10))
        i = (i == smx_top_bin ? 0 : i+1);
    ...
}
```

This is an example of bin scanning in the idle task. smx_HeapBinScan() is called once per pass through IdleMain() and 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, then i is cleared and scanning starts over at bin 0.

If the heap has 20,000 free chunks 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() 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 UG Heap Management chapter.

smx_HeapBinSeed

BOOLEAN smx_HeapBinSeed (u32 num, u32 bsz)

Type SSR

Summary Gets a big enough chunk 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.

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 with a specified block size, bsz. The bin is not specified because it depends upon the chunk size, which in turn depends upon debug mode being ON or OFF. If ON, debug chunks will be generated; if OFF inuse chunks will be generated. smx_BinSeed() shares internal subroutines with smx_HeapMalloc() and smx_HeapFree() and thus returns the same errors that they do.

smx_BinSeed() calculates the necessary chunk size for bsz and debug mode, multiplies it by num, and malloc's a big-enough chunk from the heap for that much space. It then splits the chunk into num chunks, physically links them together into the heap. Debug information is loaded into each chunk if debug mode is ON. cmerge mode is turned OFF, the new chunks are freed to their bin, cmerge is restored, and TRUE is returned.

This service, combined with monitoring bin populations, may be a good way to maintain effective bin populations. See the UG Heap chapter.

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

Example

```
u32 GetBlocksReady(u32 area, u32 bsz);
{
    u32 num = area/bsz;

    if (smx_HeapBinSeed(num, bsz);
        return num;
    else
        return 0;
}
```

This function gets as many chunks for block size, bsz, as it can within the specified area, and returns the number put into the bin for the corresponding chunk size. It returns 0 if bin seeding fails.

smx_HeapBinSort

BOOLEAN smx_HeapBinSort (u32 binno, u32 fnum)

Type	SSR	
Summary	Sorts bin's free chunk list by increasing chunk size.	
Parameters	binno	Bin number.
	fnum	Number of chunks to sort per run.
Returns	TRUE	Sort has been completed, was not needed, or error found.
	FALSE	Sort not complete.
Errors	SMXE_INV_PARM	fnum is 0.

Descr

This service is used to put chunks in order, by increasing size in large bin free lists. It is an SSR, so that it cannot be interrupted by another heap service during a run. A run consists of *fnum* loops sorting fnum chunks, or more, so higher priority tasks can preempt between runs. fnum is chosen so that higher priority tasks do not miss their deadlines. Of course the smaller that fnum is, the longer bin sorting will take.

smx_HeapMalloc() and the other heap allocation services take the first large-enough chunk from a large bin. If the bin's free list is sorted, this will be the *best fit chunk*, which is optimum to reduce fragmentation; a sorted bin also improves average allocation times for the smaller chunks in a large bin.

If binno is less than or equal to the top bin number and its bin sort map, smx_bsmap, bit is ON, that bin is sorted. If binno is greater than the top bin number, the smallest bin selected by bsmap is chosen. Calling smx_HeapBinSort() repetitively results in sorting that bin, then sorting the next larger bin, etc. until all bins have been sorted. For each bin, FALSE indicates to continue calling smx_HeapBinSort() and TRUE indicates to stop, either because the job is done or an error has been found. Hence, the binno bin will be sorted or the smallest bin will be sorted, depending upon binno vs. the top bin number.

The bsmap bit for a bin is set if smx_HeapFree() puts a chunk at the end of the bin's free list. This happens when the chunk is larger than the first chunk in the bin. Otherwise the chunk is put at the start of the bin's free list, in which case no sort is needed 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.

A bin's bsmap bit is reset when a sort begins for the bin. 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.

The sort algorithm is a *bubble sort with last turtle insertion*. The last turtle is the last chunk in the bin free list that may be smaller than a chunk ahead of it. When starting a bin sort, it will be the last chunk put at the end of the bin by the last free, if any. As a run progresses, if it finds a chunk larger than the last turtle, it puts the last turtle ahead of it and the chunk that was before the last turtle becomes the new last turtle. After k passes through the free list, the last k chunks are guaranteed by the bubble sort to be sorted, but due to last turtle moves, more than k chunks may actually be sorted. Each pass ends with the current last turtle. When no chunks have been moved during a pass, the sort is complete.

Notes

(1) Heap sorting need not be perfect. The combination of chunks in a bin, at a given time, is statistical. Hence 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

```
void smx_HeapManager(void)
{
    smx_HeapBinSort(smx_top_bin + 1, 4);
    …
}
void IdleTaskMain(void)
{
    smx_HeapManager();
    …
}
```

Here, smx_HeapBinSort() will check smx_bsmap to find the smallest large bin that needs sorting. If smx_bsmap == 0, it will abort and return TRUE. Otherwise it will start a sort on the selected bin and do a run of 4 chunks. The same bin will continue to be sorted, 4 chunks at a time, on each idle pass. This bin is remembered between runs by *smx_csbin*. When the bin is sorted, smx_HeapBinSort() sets smx_csbin to -1 in order to enable a new bin to be selected on the next pass of idle and it returns TRUE. This process continues, automatically sorting bins, as needed. Assuming that idle runs frequently, it should keep all large bins adequately sorted. If not, chunks per run can be increased or the above code could be moved to a higher priority task.

Note: smx_HeapManager() also performs auto merge control, heap scanning, and heap bin scans.

smx_HeapCalloc

void* smx_HeapCalloc (u32 num, u32 sz)

Type SSR

Summary Allocates space for an array of num elements of sz bytes from the heap and clears all elements.

Compl smx_HeapFree()

Parameters

Parameters	num	Number of elements.
	sz	Size of each element, in bytes.

Returns

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. This service uses the malloc() subroutine. Hence, it reports the same errors and returns the same result as smx_HeapMalloc().

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, SMX_PK_PARM par)

Type	SSR	
Summary	Returns the information specified by par concerning a chunk in the heap, given a pointer to either the chunk or the block in it.	
Parameters	vp	Chunk or block pointer.
	par	What to peek at.
Returns	value	Value of par.
	0	Value, unless error.
Errors	SMXE_INV_PARM	Invalid parameter.

Descr

Used to return information about heap chunks. vp is the chunk or block address. If it is out of heap range or is not 4-byte aligned, SMXE_INV_PARM is reported, and 0 is returned.

The parameter, par, is of type SMX_PK_PARM. Permitted values are:

SMX_PK_BINNO	Chunk bin number (0 if not free, dc, or tc).
SMX_PK_BP	Data block pointer from cp (0 if free).
SMX_PK_CP	Chunk pointer from bp (0 if free).
SMX_PK_NEXT	Address of next chunk in the heap.
SMX_PK_NEXT_FREE	Address of next chunk in this bin (0 if last chunk, dc, tc, or not free).
SMX_PK_ONR	Chunk owner (0 if not debug chunk).
SMX_PK_PREV	Address of previous chunk in heap.
SMX_PK_PREV_FREE	Address of previous chunk in bin (0 if first chunk, dc,tc, or not free).
SMX_PK_SIZE	Chunk size.
SMX_PK_TIME	Time chunk allocated (0 if not debug chunk).
SMX_PK_TYPE	Chunk type (free == 0, inuse == 1, debug == 3).

HeapChunkPeek() returns 0 and reports SMXE_INV_PARM, 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, except par == SMX_PK_CP, in which case it must be a valid data block pointer.

This service is an SSR. Using it 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. As shown above, smx_HeapChunkPeek() returns 0 in such a case. It usually is advisable to read the chunk type first to make sure that the expected chunk information is actually available. It also is advisable to check that the return value is not 0 before using it, except in the cases of bin number and type, where 0 returns are valid.

Example

```
u8* bp;
CCB_PTR cp;
int time = 0;
#define DEBUG 3

if (cp = (CCB_PTR)smx_HeapChunkPeek(bp, SMX_PK_CP))
    if (smx_HeapChunkPeek(cp, SMX_PK_TYPE) == DEBUG)
        time = smx_HeapChunkPeek(cp, SMX_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)

Type SSR

Summary Adds a memory extension to the heap.

Parameters

xsz	Extension size, in bytes.
xp	Extension pointer.

Returns

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

Errors SMXE_INV_PARM 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 ADAR or from other memory, but it must be above the current heap. If not, smx_HeapExtend() reports SMXE_INV_PARM, 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 smx_heap HCB is updated and TRUE is returned.

Example

```
#define HEAP_EXT  4096
u8*  xp;
BOOLEAN  ok;

if (smx_errno = SMXE_INSUFF_HEAP)
{
    xp = sb_DARAlloc(sb_adar, HEAP_EXT, 8)  /* 8-byte align */
    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. Note that if the ADAR allocation fails, HeapExtend() will also fail, because xp will be NULL.

smx_HeapFree

BOOLEAN smx_HeapFree (void* bp)

Type SSR

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

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

Parameters bp Pointer to block to free.

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_PARM	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/C++ standard. bp is converted to its corresponding chunk pointer, cp. If cp is out of range or not 8-byte aligned, smx_HeapFree() reports SMXE_INV_PARM and returns FALSE.

If the chunk is already free, the same occurs, except SMXE_HEAP_ERROR is reported. This is done to help detect double frees. However it is not 100% effective because the chunk may have already been reallocated, in which case it would not be free, and this service would free it again, causing a difficult error to find.

Next, the forward and backward links of the chunk are range-checked. If either is out of range, HeapFree() stops, reports SMXE_INV_CCB, and returns FALSE. This is another way that a double free might be detected — i.e. if the chunk were freed, merged with a lower chunk, allocated, and then the old pointers were overwritten with data (since they would be in the new data block).

The above tests and errors are enabled by SMX_HEAP_SAFE (xcfg.h), which can be turned for speed.

If merging is enabled (smx_heap.mode.cmerge == ON), HeapFree() checks if the lower chunk is free and merges it if it is, then it checks if the upper chunk is free and merges it, if it is. Chunks to be merged, except dc or tc, are removed from their bins before merging them. Then the bin for the final free chunk, unless it is dc or tc, is found and the chunk is put into that bin,. If a merger was made with dc, smx_dcp is updated; if a merger was made with tc, smx_tcp is updated. Only upward mergers into dc or tc dc are permitted and these chunks are never put into bins. heap_used is reduced by the size of the freed chunk.

If chunk filling is enabled (smx_heap.mode.fill ON) a free block is loaded with the FREE_FILL pattern; dc or tc is loaded with the DTC_FILL pattern. This greatly increases the time required to free a block and should be used selectively to assist debugging.

If either smx_hsp or smx_hfp was pointing at the freed chunk and it was merged with a lower chunk, the scan pointer is backed up to the new chunk. If a chunk is put at the end of a large bin, the smx_bsmap bit for the bin is set. This indicates 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, uses it, then frees it back to the heap.

smx_HeapInit

BOOLEAN smx_HeapInit (u32 sz, u8* hp)

Type SSR

Summary Initializes the heap. Automatically called by the first heap allocation call or can be called directly to initialize or reinitialize the heap.

Parameters

sz	Size of heap, in bytes.
hp	Heap pointer. If NULL, heap is allocated from ADAR.

Returns

TRUE	Heap has been initialized or was already initialized.
FALSE	Heap not initialized due to error.

Errors

SBE_INSUFF_DAR	Insufficient ADAR.
SMXE_INV_PARM	Invalid parameter or ADAR allocation failed.

Descr Like other smx objects, the heap is automatically initialized the first time that it is used. In this case, smx_HeapInit(sz, hp) is called by the first call to a heap allocation service. This is done so that the heap can be used by C++ initializers. In that call, sz = HEAP_SPACE and hp = HEAP_ADDRESS, which are defined in acfg.h. If HEAP_ADDRESS is NULL, then space for the heap is allocated from ADAR. Otherwise, the heap is started at hp. If sz < 32[1] or allocation from ADAR fails, smx_HeapInit() reports SMXE_INV_PARM, and returns FALSE. smx_HeapInit() can also be directly called before the first heap allocation. This may be preferable in a system where there are no C++ initializers requiring the heap.

smx_HeapInit(sz, hp) initializes the heap if smx_heap.mode.init is OFF, else it returns TRUE and nothing is done. (smx_heap is the heap control block, HCB.) 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. After the heap has been initialized, smx_heap.mode.init is set to ON. This prevents initialization from accidentally occurring twice. smx_heap.mode.init must be OFF before directly calling smx_HeapInit().

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. dc is a free chunk, which initially contains HEAP_DC_SIZE (acfg.h) bytes. tc is a free chunk, which initially contains the remaining free space of the heap = HEAP_SPACE – HEAP_DC_SIZE - 24. dc normally is much smaller than tc; it is the source for SBA chunks. tc is the source for larger chunks.

The heap is controlled by smx_heap, the static heap control block (HCB). smx_HeapInit() loads smx_heap.pi = sc and smx_heap.px = ec; it sets the cbtype and initializes the mode field so that *cmerge* and *fill* flags are turned OFF; *amerge*, *init*, *hs_fwd*, and *bs_fwd* flags are turned ON. It also clears the smx_bin[] free list pointers and the bin map, smx_bmap, so all bins are empty. It loads smx_bin_sizes[] from SMX_HEAP_BIN_SIZES (xcfg.h), and initializes the other heap variables. For the debug version (SMX_BT_DEBUG defined) dc

[1] after being adjusted to next lower multiple of 8, if necessary

and tc are filled with the SMX_HEAP_DTC_FILL (xcfg.h) pattern. This is not done for the release version for faster boot time.

Examples

```
smx_HeapSet(SMX_ST_INIT, OFF);
smx_HeapInit(0x8000, 0x20004000);

smx_HeapSet(SMX_ST_INIT, OFF);
smx_HeapInit(0x8000, 0);
```

The first example shows creating a 32KB heap at memory location 0x20004000. The second example shows creating a 32KB heap in ADAR. Note that the smx_heap.mode.init flag is turned OFF first, in both cases.

smx_HeapMalloc

void* smx_HeapMalloc (u32 sz)

Type SSR

Summary Allocates a block of at least sz bytes from the heap.

Compl smx_HeapFree()

Parameters sz Size of block to allocate, in bytes.

Returns

bp	Block pointer.
NULL	Insufficient space or error.

Errors

SMXE_INV_PARM	Invalid parameter.
SMXE_INSUFF_HEAP	Insufficient space in heap.

Descr Allocates a block of at least sz bytes from the heap. Calls smx_HeapInit() if the heap is not already initialized (i.e. smx_heap.mode.init == ON.) A *chunk* is allocated from the heap to contain the block. If debug mode is OFF, an *inuse* chunk is allocated and the block within it will be 8-byte aligned. If debug mode is ON, a *debug* chunk will be allocated and the block within it will be 4-byte aligned if SMX_HEAP_FENCES (xcfg.h) is odd, or 8-byte aligned if SMX_HEAP_FENCES is even. The minimum block size that can be allocated is 16-bytes. The block size can be larger than sz, if an exact-fit chunk was not found.

If sz is 0, SMXE_INV_PARM is reported and NULL is returned. This test is enabled by SMX_HEAP_SAFE (xcfg.h), which can be turned off for speed. 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 will be adjusted to 32. Hence, there is no problem with allocating odd sizes and they will be automatically adjusted to large enough sizes. However, it should be noted that no space savings results from such size requests.

Basically, allocation operates as follows: a small chunk comes from the right-size bin in the small bin array (SBA) or from the donor chunk (dc), if the SBA bin is empty. If dc is not big enough, the chunk comes from the first larger occupied bin, or from the top chunk (tc) if no upper bin is occupied. A large chunk comes from an upper bin or from tc. See the UG Heap Chapter for a more details concerning how heap allocation works. If the first big-enough chunk found it too big, it is split and the remant is put into a bin for its size. The remnant must be at least MIN_FRAG (xheap.c) bytes, or no split is made.

The smallest block that may be returned is sz, if sz is a multiple of 8, or the next multiple of 8, if not. This is the *adjusted size*, *adj_sz*. The largest block that may be returned is adj_sz + MIN_FRAG. The latter is defined to prevent excessively small chunks due to splitting. As delivered, it is 32 + CHK_OVH (chunk overhead.)

If debug is ON, a debug chunk is allocated instead of an inuse chunk. It consists of adding a chunk debug control block (CDCB) to the start of the chunk and adding fences before and after the allocated data block. Heap fences are word-size and their pattern is defined by SMX_HEAP_FENCE_FILL (xcfg.h). The number of fences above and below is defined by SMX_HEAP_FENCES (xcfg.h) A debug chunk can be much larger than the data block it contains and much larger than an inuse chunk with the same-size data block.

The size of the allocated chunk is added to *smx_heap_used* and the high-water mark, *smx_heap_hwm*, is increased if heap_used is larger than it. If smx_heap.mode.fill is ON, the SMX_HEAP_DATA_FILL (xdef.h) pattern is written into all words of the data block.

If allocation fails, NULL is returned and SMXE_INSUFF_HEAP is reported. This is a good reason for always checking the return value before using it.

If either bin scan pointer, smx_bsp or smx_bfp, was pointing at the chunk allocated, the bin scan is restarted. Also the smx_bsmap bit for the bin is set, which results in the bin sort being restarted.

See the UG Heap Chapter for discussion of the eheap allocation algorithm and heap recovery methods. See the UG Heap Maintenance chapter for discussion of customizing a heap for better performance.

Example

```
void* bp;

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

In this example, a block of 208 bytes is allocated from the heap, since 204 is not a multiple of 8. The block could be larger if an exact-fit chunk could not be found. When no longer needed, the block of memory is released back to the heap.

smx_HeapPeek

u32 smx_HeapPeek (SMX_PK_PARM par)

Type SSR

Summary Returns information concerning the heap mode.

Compl smx_HeapSet()

Parameters par What to peek at.

Returns value Value of par.
0 Value, unless error.

Errors SMXE_INV_PARM Invalid parameter.

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

SMX_PK_AUTO	Automatic chunk merge control is enabled.
SMX_PK_BS_FWD	Bin scan forward.
SMX_PK_DEBUG	Allocate debug chunks.
SMX_PK_FILL	Fill blocks, fences, dc, and tc with appropriate patterns.
SMX_PK_HS_FWD	Heap scan forward.
SMX_PK_INIT	Heap has been initialized.
SMX_PK_MERGE	Merge chunks, when freed.
SMX_PK_USE_DC	Allocation from donor chunk is enabled.

HeapChunkPeek() returns 0, and reports SMXE_INV_PARM, if par is not one of the above. Otherwise, it returns the value of the mode (ON or OFF). This service is an SSR. Using it is highly recommended over directly reading heap modes, which may result in incorrect readings, due to preemption by other tasks.

Example

```
if (smx_HeapPeek(SMX_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 bsz)

Type SSR

Summary Allocates a new size block from an existing heap block. Preserves existing contents and conforms to the ANSI C/C++ Standard.

Compl smx_HeapFree()

Parameters

cbp	Pointer to block to reallocate.
bsz	New block size.

Returns

nbp	New block pointer.
NULL	Insufficient space or error.

Errors Same as smx_HeapMalloc() and smx_HeapFree().

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

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

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[2]. The new free chunk is merged with the chunk after[3], 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 bsz[4]. 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

[2] There is a limitation on chunk splitting. See discussion in UG Heap Chapter, chunk splitting section.
[3] When discussing chunks, “before” and “after” or “lower” and “upper” refer to physical chunk positions.
[4] See discussion in smx_HeapMalloc().

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 allocation sub-function 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[5], 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, realloc() fails, 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 assure 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 */
/* need 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.

smx_HeapRecover

BOOLEAN smx_HeapRecover (u32 sz, u32 fnum)

Type Function.

Summary Tries to find enough free space for sz block comprised of adjacent free chunks.

Parameters

sz	Block size needed.
fnum	Number of chunks to scan per run.

Returns

TRUE	Chunk available to allocate.
FALSE	No chunk found.

[5] 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.

Errors SMXE_INV_PARM Invalid parameter

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() scans the lower heap, from sc to dc for small chunks or the upper heap from the chunk after dc to ec for large chunks. It searches for adjacent free chunks to merge. If a big-enough chunk can be formed by merging, it removes the free chunks (except dc and tc) from their bins and merges them. If the merged chunk is not dc nor tc, it puts the merged chunk into its proper bin, else it updates smx_dcp or smx_tcp, then returns TRUE.

This service does not merge chunks that it cannot use nor that it does not need. cmerge mode is ignored. If successful, smx_HeapRecover() should be followed by retrying the allocation that failed. Returns FALSE if a big-enough chunk is not found, or if sz or fnum are 0. Searching for small chunks stops at the end of the heap section (at dc or tc), if nothing is found.

This service is implemented as a function instead of an SSR, in order to be faster. It is called once per use and it is designed to allow preemption, every *fnum* chunks, so higher-priority tasks or LSRs can preempt and run. It locks the current task during each scan and unlocks the current task after each scan is complete, unless the current task is already locked. If fnum 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 lower or upper heap is reached. If a big-enough free space is found, the current task remains locked until the chunks to be merged have been removed from their bins, merged into the new big-enough chunk, and the new chunk has been put into a bin, unless it is dc or tc.

LSRs are turned OFF after the current task is locked and they are turned ON before it is unlocked, unless LSRs were already locked. This is because heap services can be called form LSRs.

Example

```
void* bp;
TCP_PTR StoppedTask;

void ProcessTaskMain()
{
   while (1)
   {
      if (bp = smx_HeapMalloc(1000))
      {
         /* process data using bp */
         smx_HeapFree(bp);
      }
      else
         break;
   }
```

```
    smx_TaskStartPar(RecoveryTask, 1000);
    StoppedTask = self;
}

void RecoveryTaskMain(u32 size)
{
    if (smx_HeapRecover(size, 100))
        smx_TaskStart(StoppedTask);
    else
        /* extend heap or stop all tasks using heap and reinitialize heap */
}
```

In the above example, if smx_HeapMalloc() fails in ProcessTask, RecoveryTask is started with the needed size as a parameter, the ProcessTask's handle is put into StoppedTask, and ProcessTask stops. RecoveryTask runs at a different priority and may run next, or not. When it does run, it calls smx_HeapRecover(), which tests 100 nodes, at time, then unlocks RecoveryTask so that higher priority tasks can preempt and run. If a big-enough free chunk is formed, RecoveryTask restarts ProcessTask, which tries again. If a big-enough free chunk is not found, ProcessTask remains stopped while other recovery techniques are tried.

smx_HeapScan

BOOLEAN smx_HeapScan (CCB_PTR cp, u32 fnum, u32 bnum)

Type SSR

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.

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_PARM	Invalid parameter.

Descr smx_HeapScan() is intended to perform continuous forward heap scans and to fix or report heap problems that it finds. Normally it is called once per pass of the idle task and forward scans fnum chunks. It is an SSR, so it cannot be interrupted by another heap service during a scan.

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 scan starts from *smx_hsp* (heap scan pointer). smx_hsp is set to NULL by smx_HeapInit() and when a heap scan is completed. As a result, the scan will begin at the start of the heap the next time smx_HeapScan() is called. At the end of each run, smx_hsp points to the next chunk to scan. Repetitively calling smx_HeapScan() with cp == NULL, results in forward scanning through the heap, fnum chunks at a time, until the end of the heap is reached. Then TRUE is returned.

For each chunk, its forward link (fl) is heap-range checked and also checked that it points after the current chunk. If not, an attempt is made to fix fl, using the chunk's size, if it is a free or debug chunk (inuse chunks have no size field). If this fails, then smx_heap.mode.hs_fwd is turned OFF, the smx_hfp (heap fix pointer) is set to the end of the heap, and FALSE is returned. As a consequence, the next time smx_HeapScan() is called, it will scan backward bnum chunks, per run, fixing broken fl's, as it goes, until it reaches smx_hsp. Normally only the one broken fl will be fixed – i.e. the one at smx_hsp. Then smx_heap.mode.hs_fwd is turned back ON and FALSE is returned. The next time smx_HeapScan() is called, the forward scan will resume.

Continuing the forward scan, the back link + flags (blf) of the next chunk is checked. If it is broken, the back link is fixed and its flags are restored. For a free or debug chunk, sz is checked and fixed if wrong. 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, and the scan continues.

Whenever a fix is made, SMXE_HEAP_FIXED is reported and the scan continues. FALSE means to continue and TRUE means to stop. If free() preempts between runs and merges the chunk pointed to by smx_hsp or smx_hfp with a lower free chunk, it backs up smx_hsp or smx_hfp to point to the new chunk. Malloc() operations do not affect these pointers.

If the backward scan finds a broken back link before it reaches smx_hsp, then it is not possible to fix the broken forward link at smx_hsp. So, instead, the gap is bridged from smx_hsp to smx_hfp and SMXE_HEAP_BRKN is reported. This is done by setting smx_hsp->fl = smx_hfp and smx_hfp->flb = smx_hsp + flags. The bridge allows the scan to finish and may allow the system to limp along, but stronger measures are needed. See UG Heap Management chapter for more information on heap scanning.

Notes (1) Because it is expected to run frequently, smx_HeapBScan() makes no entries in the event buffer, other than those due to reported errors or fixes.

Example

```
void IdleMain(void)
{
    ...
    smx_HeapScan(NULL, 2, 100);
    ...
}
```

This example shows heap scanning in the idle task. smx_HeapScan() is called once per pass through IdleMain() and will continuously scans 2 chunks, per run, starting over when it reaches the end of the heap. It will fix what it can and report what it can't.

If the heap has 200,000 chunks it will take 100,000 passes to scan. This might be too often; if slowed down to once per tick, it would take 1000 seconds (about 17 minutes) to complete a pass. This is probably fast enough. Note that a backward scan will cover 100 chunks at a time. This is because the backward scan is both faster and more urgent.

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 UG Heap Management chapter.

smx_HeapSet

BOOLEAN smx_HeapSet (SMX_ST_PARM par, u32 val)

Type SSR

Summary Sets the specified heap mode to ON or OFF.

Compl smx_HeapPeek()

Parameters

par	Parameter to set.
val	Value to set.

Returns

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

Errors SMXE_INV_PARM Invalid parameter

Descr Used to control heap modes. par is of type SMX_ST_PARM. 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.
SMX_ST_USE_DC	Use donor chunk control.

and the available values are ON and OFF. These modes are discussed in detail in UG sections. Briefly: SMX_ST_AUTO enables automatic control of chunk merge (cmerge) mode implemented in the idle task. SMX_ST_DEBUG controls debug mode, which causes allocations to create debug chunks. SMX_ST_FILL controls fill mode, which enables filling blocks with patterns, when allocated or freed. It also enables filling dc and tc with patterns. SMX_ST_MERGE control cmerge mode, which applies to free operations. SMX_ST_USE_DC controls whether allocations come from the donor chunk, if the selected SBA bin is empty.

This service is an SSR. Using it is highly recommended over directly setting internal heap modes, which may result in incorrect settings due to preemption of the current task. If par is not recognized, returns FALSE and reports SMXE_INV_PARM. val is not checked: 0 == OFF, 1 == ON.

Example

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

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

smx_HT

smx_HT

void   smx_HT_ADD (VOID_PTR handle, const char *name)
void   smx_HT_DELETE (VOID_PTR handle)
VOID_PTR   smx_HTGetHandle (char *name)
const char *   smx_HTGetName (VOID_PTR handle)

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

Summary Add and delete entries to the handle table (HT). Query HT for handles or names. Used by smxAware.

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

Returns none

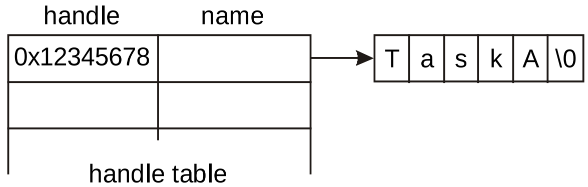
Errors	SMXE_HT_DUP	Duplicate entry
	SMXE_HT_FULL	Handle table full

Descr smx_HT_ADD() adds an entry for handle to the handle table. smx_HT_DELETE() deletes the entry. Most smx create calls automatically add an entry to the handle table, and most delete calls delete the entry. These macros are generally used to give names to objects which have no control blocks, such as ISRs and LSRs. If you manually add a name to the handle table, you must manually delete it before deleting the object.

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() 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), scans to see if name is already in the handle table and reports SMXE_HT_DUP if it is.

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_PTR 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 = smx_HTGetName(task);
    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 RM smx_SysPseudoHandleCreate(). In appl_init(), smx_HT_Name() assigns “MyISR” to this handle and creates an entry in HT. smx_TaskCreate() automatically creates an HT entry for TaskA.

The print_report() function is able to get the names from the task and ISR handles passed to it 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() automatically deletes the TaskA entry in HT and smx_HT_DELETE() is called to delete the MyISR entry in HT.

smx_ISR

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, incrementing smx_srnest, and switching to SS. Some processors (e.g. Cortex-M3) do all of these automatically and smx_ISR_ENTER() is a NOP. Others require all of these to be done (e.g. some ARM's). In addition, some processors necessitate using assembly macros; others allow C macros. Implementation of smx_ISR_ENTER() is a complex subject. See UG Interrupt Handling and TG for your processor and tool suite.

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. This is the ideal case.

Example 2

```
        EXTERN  AnISRC
        PUBLIC  AnISR
AnISR:
        smx_ISR_ENTER
        LDR     r0, =AnISRC
        MOV     lr, pc
        BX      r0          ;call AnISRC()
        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
    ; call dispatcher
    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 figures out which ISR to call, then calls it, such as AnISR() 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 ISR. Binds interrupt service routine to 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(). 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 if the LSR queue (lq) is not empty, branches to the prescheduler, which calls the LSR scheduler. If lq is empty, 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. When all LSRs have run, the prescheduler determines whether to call the task scheduler or to return to the current task.
Examples	See smx_ISR_ENTER().

smx_LSR

smx_LSR_INVOKE

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

Types **smx_LSR_INVOKE** unrestricted C macro/function for use from ISRs and LSRs
smx_LSRInvoke SSR for use from tasks

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

Parameters

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

Returns

TRUE	LSR invoked.
FALSE	Error.

Errors SMXE_LQ_OVFL

Descr Places the LSR address, lsr, followed by the parameter, par, into the LSR queue, lq. If lq is full, reports SMXE_LQ_OVFL and aborts. If smx_LSRInvoke() is called from a task, lsr will preempt and run immediately, unless LSRs are off (see below). If smx_LSR_INVOKE() is called from an ISR or an LSR, lsr will run after all LSRs ahead of it in lq have run. smx_LSR_INVOKE() converts to smx_LSRInvokeF().

Assembly versions of smx_LSRInvokeF() do not exist, but can be derived from disassembly of the C versions, then optimized, if desired.

LSR Main void lsr(u32 par)

When lsr is dispatched, par is accessible to it as a normal C function parameter.

Notes:

(1) The interrupt enable state is saved and disabled at the start, then restored at the end.

(2) Use only the macro version in ISRs. Do not call the SSR smx_LSRInvoke() from ISRs. The macro does not return a value because it is for use from ISRs; it only fails if the LSR queue overflows; and there is nothing the ISR can do to retry since LSRs will not run until the ISR completes.

(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.

Example

```
SCB_PTR  send_done;

void send_main(void)
{
    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_INF);
}

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

void send_LSR(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 waits. When the output device needs the next character, it interrupts to cause send_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 restart the process.

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 Bare C macro

Summary Inhibits LSRs from running.

Compl smx_LSRsOn()

Parameters none

Returns none

Errors none

Descr Used in combination to prevent LSRs from running. This makes the code atomic because an interrupt cannot cause a preemption. Particularly useful for code directly accessing or altering smx control blocks or other globals and to prevent undesirable preemptions. Effect is similar to smx_TaskLock(), except that locking does not prevent LSRs and SSRs from running.

Example

```
void atask_main(void)
{
    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()
Parameters	none
Returns	TRUE
Errors	none
Descr	Re-enables LSRs.
Example	See above.

smx_Msg

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	
SMXE_INV_PRI	pri > SMX_MAX_PRI
SMXE_INV_XCB	

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 msg will still be requeued if in a valid exchange queue.

Example

```
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);
    smx_MsgXchgClear(xa);
}
```

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, u16 clrsz)

Type SSR

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

Compl smx_MsgRel()

Parameters

pool	Pool to get message body from.
bpp	Pointer to message body (block) pointer. NULL if none.
clrsz	Number of bytes to clear from the start of message body.

Returns

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

Errors SMXE_INV_PCB
SMXE_OUT_OF_MCBS

Descr Gets a block from the specified block pool for use as the message body and an MCB from the MCB pool, initializes the MCB and links it to the message body. Clears the first clrsz bytes of the message body up to its size and loads the address of the message body into bpp, unless it is NULL. bpp is intended to be used to load data into the message body. Returns the message handle. If pool is invalid or if out of MCBs, aborts, returns NULL, and bpp is not changed. The current task or current LSR becomes the message owner.

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*  mbp;
    MCB_PTR  msg;

    msg = smx_MsgGet(pool, &mbp, 4);
    /* load message using mbp */
    return msg;
}
```

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

smx_MsgMake

MCB_PTR   smx_MsgMake (PCB_PTR pool, u8 *bp)

Type	SSR	
Summary	Makes a message from a bare block.	
Compl	smx_MsgUnmake()	
Parameters	pool	Base pool of the block. NULL if none
	bp	Block pointer.
Returns	msg	Handle of message obtained.
	NULL	Insufficient resources or error.
Errors	SMXE_OUT_OF_MCBS	
	SMXE_INV_PARM	bp == NULL

Descr Makes a message from a bare block. Gets MCB from MCB pool, initializes it, and returns its handle. The pool pointer is stored in the MCB (NULL if no pool).

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. Bare blocks can be statically defined, obtained from a base pool, DAR, heap, or ROM, or any other source.

Example:

```
PCB         in_pool;
XCB_PTR  in_xchg;

void inISR(void);
{
    u8 ch;
    u8 *mbp, *dp;

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

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

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

inISR() 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, inLSR() is invoked. inLSR() 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_PARM par)

Type SSR

Summary Returns the current value for the parameter specified.

Parameters

msg	Message to peek at.
par	Argument to return.

Returns

value	Value of par.
0	Value, unless error.

Errors

SMXE_INV_MCB
SMXE_INV_PARM
SMXE_BROKEN_Q
SMXE_UNKNOWN_SIZE

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

SMX_PK_BP	Body pointer.
SMX_PK_ONR	Owner.
SMX_PK_NEXT	Next msg in queue. NULL, if none.
SMX_PK_PRI	Priority.
SMX_PK_POOL	Pool.
SMX_PK_REPLY	Reply field. 0 if mcb.rpx = 0xFFFF (= no reply).
SMX_PK_SIZE	Size.
SMX_PK_XCHG	Exchange where msg is waiting. 0, if none, or broken queue.

If there is no pool, SMX_PK_SIZE will return 0 and report SMXE_UNKNOWN_SIZE. This is because message size is stored in the pool PCB. Hence, if a message is made from a free 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 = process_msg(msg);
    reply = (XCB_PTR)smx_MsgPeek(msg, SMX_PK_REPLY);
    *mbp = pass;
    smx_MsgSendPR(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, u32 timeout)

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 body (block) pointer. NULL if none.
timeout	Timeout in ticks.

Returns

msg	Message handle.
NULL	Error or timeout.

Errors

SMXE_INV_XCB	
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 it 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 ct on xchg for timeout period. ct is enqueued in priority order. If a message is sent to xchg before the timeout elapses, 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 unless the message priority is not less than SMX_MAX_PRI. It first changes the normal priority of ct to that of the message. It then changes the current priority of ct to that of the message, unless ct owns a mutex. If ct owns a mutex, current priority is changed up, but not down in order to preserve priority promotion by the mutex. (Moving current priority down to normal priority will occur when ct has released all mutexes.) Requeues ct in the ready queue if its priority has changed. Note: If ct's 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, 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.

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(void)
{
    u8  *mbp;
    while (1)
    {
        if (msg = smx_MsgReceive(in_xchg, &mbp, TMO))
            /* process msg using mbp */
        else
            /* do something else */
    }
}
```

In the above example, task gets msg from the in_xchg and processes it, using mbp. task will wait up to TMO ticks, then there is no message, it will do something else. This process continues indefinitely.

smx_MsgReceiveStop

void smx_MsgReceiveStop (XCB_PTR xchg, u8 **bpp, u32 timeout)

Type limited SSR — tasks only

Summary Same as smx_MsgReceive() except that 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 body (block) pointer. NULL if none.
timeout	Timeout in ticks.

Errors

SMXE_INV_XCB	
SMXE_INV_PARM	bpp points to a location in the current 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. ct always stops, then restarts instead of 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) smx_lockctr is cleared if called from a task.

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 Calls notes and restrictions.

Example

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

void task_Main(u32 msg)
{
    if (msg != NULL)
        /* process data using mbp */
    else
        /* do something else */
    smx_MsgReceiveStop(input, &mbp, TMO);
}
```

The above example is equivalent to the example shown before 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)

Type SSR

Summary Releases a message obtained by smx_MsgGet().

Compl smx_MsgGet()

Parameters msg Message to delete.

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

Errors SMXE_INV_MCB

Descr Releases a message obtained by smx_MsgGet(). Releases the block used for the message body and the MCB back to their pools. Clears clrsz bytes up to the end of the block. Returns TRUE if successful. Fails and returns FALSE if msg is not valid. This can include an invalid handle or invalid pool or mbp fields in the MCB. The operation also fails if the message is not owned by the current task. The latter is done for safety to prevent a task , which 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.

Will first dequeue a message if it is in a queue. This allows a broadcast task to release a message it sent to a broadcast exchange, since it still owns the message.

Note 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 (const TCB_PTR task)

Type SSR

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

Parameters task Task whose messages are to be released.

Returns Number of messages released.

Errors SMXE_INV_TCB

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. This may be necessary because atask may own a message at the time that another task stops it.

smx_MsgSend

BOOLEAN smx_MsgSend (MCB_PTR msg, XCB_PTR xchg)
BOOLEAN smx_MsgSendPR (MCB_PTR msg, XCB_PTR xchg, u8 pri, void *reply)

Type macro and 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 if no reply is expected.

Returns

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

Errors

SMXE_INV_MCB	invalid or ct or clsr is not the owner.
SMXE_INV_XCB	
SMXE_INV_PARM	reply is not a QCB handle, nor NULL.

Descr smx_MsgSend() is a macro that calls smx_MsgSendPR() with 0 priority and NULL reply.

If xchg is a **normal exchange**, msg is enqueued in its wait queue, unless there are one or more tasks 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. The latter will probably be the norm, in most systems. Also xchg becomes the message owner. Hence, the message will not be released if the task is deleted or smx_MsgRelAll(task) is called. Note that the task making this call must be the message owner, else the call fails. This is done for safety to prevent the same message from being sent twice, except in the special case of a broadcast exchange. Note: an LSR can send a message that it does not own. This is done to allow message handles to be passed to LSRs as LSR parameters.

If xchg is a **pass exchange**, operation is the same with the addition that the receiving task assumes the message's priority. See the discussion of this above under smx_MsgReceive().

If xchg is a **broadcast exchange,** operation is quite different. Tasks are enqueued in FIFO order and all are resumed at once by MsgSend(). Each task receives the msg handle and the message body pointer is loaded into the its mbp location. However, the sending task remains the owner and the message "sticks" to the broadcast exchange, meaning that it will be "received" by all subsequent receives until replaced or released. The message can be replaced by sending another message to the xchg or it can be released by the initial sender. In the first case, the message will be automatically released; 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 UG broadcasting messages for more information.

The reply parameter allows the sender to tell the msg recipient where to reply. Usually it is an exchange handle to send a reply message, but it could be a semaphore, event group, or event queue. These are all in the QCB pool. The reply handle is stored in the MCB in the form of a 16-bit index, rpx, into the QCB pool, instead of a 32-bit handle in order to save space. If the reply parameter == NULL, then rpx is set to 0xFFFF, meaning no reply. See UG **using the reply field** for more information. If the pri parameter = SMX_PRI_NOCHG, the message priority is not changed. This allows a task to move a message from one exchange to another without changing its priority.

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)) != NULL)
    {
        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 UG client/server example for a reply example.

smx_MsgUnmake

u8 *smx_MsgUnmake (PCB_PTR *pool, MCB_PTR msg)

Type SSR

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

Compl smx_MsgMake()

Parameters

pool	Place to put pool handle if != NULL.
msg	Message to unmake.

Returns

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

Errors SMXE_INV_MCB

Descr Reverses smx_MsgMake() by converting an smx message to a bare block by releasing its MCB. Fails and returns NULL if msg is not valid. This can include an invalid handle or invalid pool or mbp fields in the MCB. The operation also fails if the message is not owned by the current task. The latter prevents a task, which no longer owns a message, from unmaking it, which is done for safety. Note: an LSR can unmake a message that it does not own. This is done to allow message handles to be passed to LSRs as LSR parameters.

Otherwise, returns the address of the data block and loads its pool handle, if any, into the user-supplied location, unless NULL. (For a base block, the code receiving the block must know its pool handle.)

Example

```
u8*  bpi;
PCB_PTR  msg_pool;
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(outLSR, (u32)msg);
}
```

```
void outLSR(u32 m)
{
    MCB_PTR msg = (MCB_PTR)m;

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

void outISR(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 outLSR() with msg as its parameter. outLSR() unmakes the message, thus loading bpi and ppi for outISR(). outLSR() then outputs the first character to start UART output. The UART interrupts each time it needs another character and outISR() provides the character until all characters have been sent. outISR() releases the block back to msg_pool, which is pointed to by ppi, in this case. In other cases, it could point to a base block pool or be NULL. If NULL, the block is not released to any pool. Regardless of how msg was formed, its message block is released to where it belongs.

Notes:

1. Using outLSR() is not essential — its functions could just as well be performed by a task, if preferred.
2. The above example implements no-copy message output.

smx_MsgXchg

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
SMXE_BROKEN_Q

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. Returns TRUE, if successful or FALSE if invalid xchg. Aborts and reports error if task or message queue is broken.

Example

```
TCB_PTR serverA, serverB;
BOOLEAN modeA;

BOOLEAN toggle_server(void)
{
    BOOLEAN pass = FALSE;

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

```
    modeA ^= 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 complete 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)

Type SSR

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

Compl smx_MsgXchgDelete()

Parameters

mode	Operating mode.
name	Name to give exchange, NULL if none.

Returns

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

Errors SMXE_OUT_OF_QCBS
SMXE_WRONG_MODE

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 QCB pool and initializes it. If allocation fails because of an invalid mode or no block is available, returns NULL. Otherwise returns the exchange handle. Creates QCB pool on first use.

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 and a pass exchange.

smx_MsgXchgDelete

BOOLEAN smx_MsgXchgDelete (XCB_PTR *xchg)

Type SSR

Summary Deletes an exchange.

Compl smx_MsgXchgCreate()

Parameters xchg Exchange to delete.

Returns TRUE Exchange deleted or already NULL.
FALSE Error.

Errors SMXE_INV_XCB
SMXE_BROKEN_Q

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, removes its name from HT, and clears its handle. Aborts and reports error if task or message queue is broken.

Example

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

    if (smx_TaskStop(serverA, INF))
        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 insures that more messages cannot be sent.

smx_MsgXchgPeek

u32 smx_MsgXchgPeek (XCB_PTR xchg, SMX_PK_PARM par)

Type SSR

Summary Returns the current value for the parameter specified.

Parameters

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

Returns

value	Value of par.
0	Value, unless error.

Errors

SMXE_INV_XCB	Invalid message exchange handle.
SMXE_INV_PARM	Invalid argument.

Notes 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)
{
    CB_PTR cb;
    u32 ctr = 0;

    smx_TaskLock();
    if ((cb = (CB_PTR)smx_MsgXchgPeek(xchg, SMX_PK_MSG)) != NULL)
        for (; cb->cbtype != SMX_CB_MCB; cb = 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. Hence, this is not an optimum way to implement this capability. It would be better to add an SMX_PK_NUM_MSG argument to smx_MsgXchgPeek().

smx_Mutex

smx_MutexClear

BOOLEAN smx_MutexClear (MUCB_PTR mtx)

Type SSR

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

Compl smx_MutexGet()

Parameters mtx Mutex to clear.

Returns TRUE Mutex cleared.
FALSE Error.

Errors SMXE_BROKEN_Q
SMXE_INV_MUCB

Descr If the mutex is owned, 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. It the owner priority has changed and it is in a queue, requeues it. Resumes all tasks waiting at mtx, with FALSE returns. Reports broken queue, if encountered and stops. Puts MUCB in its cleared state.

Normally, smx_MutexRel() is what a task should call to release a mutex it owns. smx_MutexClear() is called by smx_MutexDelete(), and it can also be used for recovery.

Example

```
MUCB_PTR mtx;

void task_main(void)
{
    mtx = smx_MutexCreate(1, PRI_HI, "mtx");
    //...
    smx_MutexClear(mtx);
}
```

smx_MutexCreate

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

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.

Returns

handle	Mutex created.
NULL	Insufficient resources or error.

Errors SMXE_OUT_OF_MUCBS

Descr If pi != 0, priority inheritance is enabled. If ceiling != 0, it specifies the ceiling priority of the mutex. These are used to avoid unbounded priority inversion of tasks. See UG Mutexes for discussion of this topic.

Example

```
MUCB_PTR mtx;

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

smx_MutexDelete

BOOLEAN smx_MutexDelete (MUCB_PTR * mtx)

Type SSR

Summary Deletes a mutex created by smx_MutexCreate().

Compl smx_MutexCreate()

Parameters mtx Mutex to delete.

Returns TRUE Mutex deleted.
FALSE Error.

Errors SMXE_INV_MUCB

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 frees and clears the MUCB and clears its handle in mtx.

Example

```
MUCB_PTR mtx;

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

smx_MutexFree

BOOLEAN smx_MutexFree (MUCB_PTR mtx)

Type SSR

Summary Makes the next waiting task the new owner or frees the mutex if none waiting, regardless of owner and nesting count.

Compl smx_MutexGet()

Parameters mtx Mutex to free.

Returns TRUE Mutex freed.
FALSE Error.

Errors SMXE_BROKEN_Q
SMXE_INV_MUCB

Descr Makes the next waiting task the new owner or frees the mutex if no task waiting. Resumes the previous owner with FALSE and adjusts its priority to the highest owned mutex priority or to normal priority, if none. The previous owner is requeued, if its priority changes. Differs from smx_MutexRel() in that smx_ct does not need to be the owner. Differs from smx_MutexClear() in that it does not clear the task wait queue of mtx.

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 a mutex. It also should be called before stopping a task that owns a mutex.

Example

```
void stop_task(TCB_PTR task)
{
    while (task->mtxp != NULL)
    smx_MutexFree(task->mtxp);
    smx_TaskStop(task, INF);
}
```

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)

Types SSR

Summary Gets mutex, if free and returns TRUE. Otherwise, ct is suspended and put into mutex's wait queue.

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

Parameters

mtx	Mutex to get.
timeout	Timeout in ticks.

Returns

TRUE	Got mutex.
FALSE	Error or timeout.

Errors

SMXE_INV_MUCB	
SMXE_LSR_NOT_OWN_MTX	Called from an LSR.

Descr When the current task gets a mutex, it becomes the owner of the mutex. 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. TRUE is returned. If the mutex has a ceiling priority that is higher than the task's current priority, the task's priority is promoted to it. 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, ct is suspended and priority enqueued in mtx's wait queue. If priority inheritance is enabled in mtx, the priority of the owner is promoted to ct->pri, if it is greater. This enables the owner to finish its operation without preemption by mid-priority tasks (to prevent unbounded priority inversion) and to release the mutex to the waiting task sooner.

This function cannot be called from an LSR, since LSRs cannot own mutexes. Attempting to do so results in an error and the call aborts without doing anything.

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

Example

```
MUCB_PTR mtx;

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

This example shows protecting a critical section of code with a mutex.

smx_MutexGetStop

void smx_MutexGetStop (MUCB_PTR mtx, u32 timeout)

Type limited SSR — task 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.

Errors

SMXE_INV_MUCB	
SMXE_OP_NOT_ALLOWED	Called from an LSR.

Descr See smx_MutexGet() for operational description. ct always stops, 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, aborts operation and returns to LSR.
(2) smx_lockctr is cleared if called from a task.

TaskMain void task_main(BOOLEAN par)

par

TRUE	Got mutex.
FALSE	Error or timeout.

Example

```
MUCB_PTR mtx;
BOOLEAN pass = 1;

pass = smx_MutexGet(mtx, tmo);
smx_TaskStartPar(taskA, 0);

void taskA_Main(BOOLEAN pass)
{
    if (pass)
    {
        /* do critical section */
        smx_MutexRel(mtx);
    }
    else
        /* startup or deal with timeout or error */
    smx_MutexGetStop(mtx, TMO);
}
```

The above example shows protecting a critical section with a mutex for a one-shot task. When first started, since par = 0, taskA attempts to get mtx and then 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 re-acquire it and stops. taskA waits up to TMO 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 ct and nesting count == 1, else if nest count > 1 just decrements it.

Compl smx_MutexGet()

Parameters mtx Mutex to release.

Returns
TRUE Mutex released.
FALSE Error.

Errors
SMXE_INV_MUCB
SMXE_LSR_NOT_OWN_MTX
SMXE_MTX_ALRDY_FREE
SMXE_MTX_NON_ONR_REL
SMXE_BROKEN_Q

Descr If mtx is already free or ct is not its owner, operation aborts and an error is generated. Similarly if mtx is invalid or this function was called from an LSR. Otherwise, decrements the mtx nesting count and if not zero, returns with TRUE. If nesting count is zero, removes mtx from ct's mutex-owned list, and, if ct does not own any other mutexes, its priority is restored to its normal priority. Otherwise smx_ct->pri is set to the highest ceiling priority or the highest waiting task priority for other mutexes owned by ct. If ct's priority changes, it is requeued in rq and test for preemption 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 its priority is promoted to mtx ceiling, if greater.

This is the function that normally should be called to release a mutex obtained with smx_MutexGet(). Two similar functions are provided for special purposes: smx_MutexClear() and smx_MutexFree(). See their descriptions.

Example See the smx_MutexGet() example.

smx_Pipe

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

Descr Sets pipe read and write pointers to the start of the pipe buffer, thus clearing the pipe. Then resumes all tasks waiting on the pipe with FALSE. 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)

Type SSR

Summary Creates a pipe.

Compl smx_PipeDelete()

Parameters ppb Pointer to pipe buffer, which points to a block allocated by the user for the pipe buffer. For best performance, this buffer should be aligned on a 32-bit or cache-line boundary and located in SRAM.
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. Pipe length is one greater than the maximum number of packets that can be stored in the pipe. This is because one cell is sacrificed to distinguish a full pipe from an empty pipe.
	name	Name to give pipe; NULL, if none.
Returns	handle	Pipe created.
	NULL	Pipe not created due to error.

Errors SMXE_INV_PARM 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. The cell size determines the maximum packet size that the pipe will accept.

Warning 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 with (width * length).

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 and its address is passed as a parameter to pipe create. Buffers can be statically defined, as shown, or obtained from a block pool, the heap, or a DAR. 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 so that data following pkt_pipe would not be overwritten.

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

Descr Deletes a pipe by resuming all waiting tasks with FALSE return values, releasing its PICB back to the PICB pool, and clearing its handle so it cannot be used again. The user must manage the pipe buffer (e.g. re-use it or release it back to its block pool or the 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 it back to the heap. This illustrates the convenience of return of the pipe buffer address from pipe delete. However, as discussed in the smx User's guide, it is not a good idea to allocate pipe buffers from the heap for pipes that are routinely created and deleted. Note, in the second function, that if pipe delete failed, ppb would be 0, heap free would fail, and close pipe would return FALSE.

smx_PipeGet

BOOLEAN smx_PipeGet (PICB_PTR pipe, void *pdst)
void smx_PipeGetPkt (PICB_PTR pipe, u8 *pdst)

Type Bare functions

Summary Gets the next packet from pipe and loads it into the buffer at pdst. First is for LSR and task usage. Second is for ISR usage.

Compl smx_PipePut functions and SSRs.

Parameters

pipe	Pipe handle. Operation is aborted if not valid.
pdst	Destination pointer to store packet. Operation is aborted if zero.

Returns

TRUE	Packet transferred.
FALSE	Packet not transferred.

Errors SMXE_INV_PARM
SMXE_INV_PICB

Descr If pipe is not empty, smx_PipeGet() copies the oldest packet from it to the buffer at pdst, advances the pipe's read pointer to the next cell, and returns TRUE; otherwise returns FALSE. Also returns FALSE if pipe is invalid or if pdst is NULL. Does not wait. smx_PipeGet() may be used in time-critical sections of user code such as in LSRs and tasks, which cannot wait. If this function is used in tasks, it must be protected from preemption, since it is not an SSR.

smx_PipeGetPkt() is for use in ISRs. It does no error checking and will not return reliable data if the pipe is empty or an error is encountered. Its use will not interfere with an interrupted complementary function operating on the same pipe, providing it is not operating on the same packet.

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 considered empty unless a full packet is present.

Example

```
PICB_PTR  pkt_pipe; /* width = 8 */

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

    while (smx_PipeGet(pkt_pipe, mb))
    {
        for (i = 0; i < 8; i++)
        {
```

```
            *out_port = mb[i];
        }
    }
}
```

In this example, an 8-byte packet is being output from the 8-byte-wide pkt_pipe, byte by byte. These packets are probably formatted messages, having a common structure. Hence, it makes sense for the task loading the pipe to deal with packets, not with a byte stream. Note that smx_PipeGet() does not wait if the pipe is empty; instead, it returns FALSE, the while statement fails, and out_pkt() exits. It must be called again to output another packet.

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.

Returns
TRUE Byte transferred.
FALSE Byte not transferred.

Errors None

Descr If pipe is not empty, returns the oldest byte in it, advances the pipe's read pointer to the next cell, and returns TRUE; otherwise returns FALSE. 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, 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 put a byte. Use smx_PipeResume(pipe) for this purpose.
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. This function might be called from an LSR that was invoked from a timer or from an ISR invoked by an interrupt.

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 the fast version of smx_PipeGet8() for multi-byte transfers, 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, 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 put a byte. Use smx_PipeResume(pipe) for this purpose.
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; it can be used to determine how many bytes to process downstream. This function might be called from an LSR that was invoked from a timer or from an ISR invoked by an interrupt (e.g. due to an empty UART output buffer).

smx_PipeGetWait

BOOLEAN smx_PipeGetWait (PICB_PTR pipe, void *pdst, u32 tmo)

Type SSR

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

Compl smx_PipePut functions and SSRs.

Parameters

pipe	Pipe handle. Operation is aborted if not valid.
pdst	Destination pointer to store packet. Operation is aborted if zero.
tmo	Timeout in ticks. If called from an LSR, tmo must be 0.

Returns

TRUE	Packet transferred.
FALSE	Packet not transferred.

Errors

SMXE_INV_PARM
SMXE_INV_PICB
SMXE_WAIT_NOT_ALLOWED

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, resumes the waiting task, and sets its return value = TRUE. If cannot get a packet and tmo > 0, the current task is suspended, until either it gets a packet or a timeout occurs. Can be used from a task or an LSR. If called from an LSR, tmo must be 0.

Notes

1. Use only with complementary functions at the other end of the pipe.
2. May be mixed with smx_PipeGetWaitStop()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. Clears smx_lockctr if called from a task and timeout != SMX_TMO_NOWAIT.

Example

```
PICB_PTR in_pipe, ctrl_pipe, data_pipe;  /* all widths = 8 */
TCB_PTR  msg_switch_task;
struct ctrl_msg
{
    u8 type;
    u8 dest;
    u8 rdg[6];
};

void msg_switch_main(void)
{
    struct ctrl_msg m;
    BOOLEAN done;

    while (smx_PipeGetWait(in_pipe, &m, 100))
    {
        if(m.dest == 1)
        {
            done = smx_PipePutWait(ctrl_pipe, &m, 100);
        }
        else
        {
            done = smx_PipePutWait(data_pipe, &m, 100);
        }
        if(!done) break;
    }
    report_problem();
}
```

In this example, the msg_switch task waits on in_pipe for a message. If a message is received, it tests the dest field to determine where to send the message. If the dest field is 1, the message is sent to the ctrl_pipe, otherwise it is sent to data_pipe. If either output pipe is full, the task will wait. In all three cases a maximum wait of 100 ticks is specified, after which a problem is reported and the task stops. Once the problem is resolved, the task can be restarted.

smx_PipeGetWaitStop

void smx_PipeGetWaitStop (PICB_PTR pipe, void *pdst, u32 tmo)

Type limited SSR — task only

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

Compl smx_PipePut functions and SSRs.

Parameters

pipe	Pipe handle.
psrc	Destination pointer to store packet.
tmo	Timeout in ticks.

Errors

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

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

Notes

1. Use only with complementary functions at the other end of the pipe.
2. May be mixed with smx_GetWaits()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. smx_lockctr is cleared if called from a task.

TaskMain void **task_main**(BOOLEAN par)

par

TRUE	Packet transferred
FALSE	Packet not transferred

Example

```
PICB_PTR key_pipe;  /* byte wide pipe */
TCB_PTR  key_task;
u8       key_buf[40];
u8       input_key(u8 key_port);
void     process_key(u8 *bp);
void     report_problem(void);
void     key_task_main(BOOLEAN got_key);

void key_task_init(void) /* initial task main function when created */
{
    key_task->fun = (FUN_PTR)key_task_main;
    smx_PipeGetWaitStop(key_pipe, (void *)key_buf, 100); /* wait for first key */
}
```

```
void key_task_main(BOOLEAN got_key)
{
    if(got_key)
        process_key(key_buf);
    else
        report_problem();
    smx_PipeGetWaitStop(key_pipe, (void *)key_buf, 100); /* wait for next key */
}

void key_LSR(u32 par)
{
    smx_PipeResume(key_pipe);
}

void key_ISR(void) /* invoked by interrupt */
{
    u8 ch;

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

In this example, key_task is a one-shot task. Note that key_task_init() performs the initial pipe get wait stop operation. Prior to this, the task entry point is changed to key_task_main(), which expects a got_key TRUE/FALSE parameter from the get operation. When the get operation in key_task_init() gets a key, it starts key_task at key_task_main(), where the key is processed and another get wait stop operation is started. The get operations wait up to100 ticks to get keys. More detail is shown below the task code: key_ISR() is the ISR that gets a key from the key port, puts it in the key_pipe and invokes key_LSR. key_LSR performs a pipe resume to awaken the waiting key task on key_pipe.

Note that it is possible that several key interrupts could occur before key_task is able to run – especially if it is a low priority task. Hence there could be several keys waiting in key_pipe. This causes no harm 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. This code could be more efficiently written for an incoming data stream, but for keystrokes it should be adequate.

smx_PipePut

BOOLEAN	smx_PipePut (PICB_PTR pipe, void *psrc)
void	smx_PipePutPkt (PICB_PTR pipe, u8 *psrc)

Type Bare functions

Summary Puts the packet from the buffer at psrc into pipe. First is for LSR and task usage. Second is for ISR usage.

Compl smx_PipeGet functions and SSRs.

Parameters

pipe	Pipe handle. Operation is aborted if not valid.
psrc	Pointer to source of packet. Operation is aborted if zero.

Returns

TRUE	Packet transferred.
FALSE	Packet not transferred.

Errors SMXE_INV_PARM
SMXE_INV_PICB

Descr If the pipe is not full, smx_PipePut() copies the packet in the buffer at psrc into it, advances the pipe's write pointer to the next cell, and returns TRUE; otherwise returns FALSE. Also returns FALSE if pipe is invalid or if psrc is NULL. Does not wait. smx_PipePut() may be used in time-critical sections of user code such as in LSRs and tasks, which cannot wait. If this function is used in tasks, it must be protected from preemption, since it is not an SSR.

smx_PipePutPkt() is for use in ISRs. It does no error checking and will not operate reliably if the pipe is full or an error is encountered. Its use will not interfere with an interrupted complementary function operating on the same pipe, providing it is not operating on the same packet.

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;

    for(i = 0; i < 10; i++)
        input(mb[i], in_port);
    smx_PipePut(msg_pipe, mb);
}
```

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_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, advances the pipe's write pointer to the next cell, and returns TRUE; otherwise returns FALSE. This is the fast version of smx_PipePut() for byte puts; 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, 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 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)  /* key received, start task via LSR */
}
```

In this example, key_ISR() is invoked by 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. For more of this example, see smx_PipeGetWaitStop().

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 usage.

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 the fast version of smx_PipePut8() for multi-byte transfers, as may occur with UARTs and other high-speed serial controllers. 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, 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[]. numx is the actual number of bytes transferred; it can be used to determine when to ask for more bytes (e.g. --numx < 3). This function might be called from an LSR that was invoked from a timer or from an ISR needing more bytes in out_pipe (assuming the ISR is using smx_PipeGet*() to get bytes to send out).

smx_PipePutWait

BOOLEAN smx_PipePutWait (PICB_PTR pipe, void *psrc, u32 tmo)

Type SSR

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

Compl smx_PipeGet functions and SSRs.

Parameters

pipe	Pipe handle. Operation is aborted if not valid.
psrc	Pointer to source of packet. Operation is aborted if zero.
tmo	Timeout in ticks. If called from an LSR, tmo must be 0, else operation is aborted.

Returns

TRUE	Packet transferred.
FALSE	Packet not transferred.

Errors
SMXE_INV_PARM
SMXE_INV_PICB
SMXE_WAIT_NOT_ALLOWED

Descr If another task is waiting to get a packet, gives the packet in the buffer at psrc to it and resumes the waiting task with TRUE, else if the pipe is not full, copies the packet into pipe and advances pipe's write pointer to its next cell. Returns TRUE, in both cases. If neither case and tmo > 0, the current task is suspended until either its packet is accepted or the timeout occurs. Can be used from a task or an LSR. If called from an LSR, tmo must be 0.

Notes
1. Use only with complementary functions at the other end of the pipe.
2. May be mixed with smx_PutWaitStop()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 != SMX_TMO_NOWAIT.

Example See smx_PipeGetWait() example.

smx_PipePutWaitStop

void smx_PipePutWaitStop (PICB_PTR pipe, void *psrc, u32 tmo)

Type limited SSR — task only

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

Compl smx_PipeGet functions and SSRs.

Parameters

pipe	Pipe handle.
psrc	Pointer to source of packet.
tmo	Timeout in ticks.

Errors SMXE_OP_NOT_ALLOWED
SMXE_INV_PARM
SMXE_INV_PICB

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

Notes

1. Use only with complementary functions at the other end of the pipe.
2. May be mixed with smx_PutWaits()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. smx_lockctr is cleared if called from a task.

TaskMain void task_main(BOOLEAN par)

par

TRUE	Packet transferred.
FALSE	Packet not transferred.

Example

```
PICB_PTR  crt_pipe;
TCB_PTR  crt_task;
u8  crt_buf1[80], crt_buf2[80];
BOOLEAN  tog;

void crt_task_init(void)
{
    crt_task->fun = (FUN_PTR)crt_task_main;
    tog = TRUE;
    smx_PipePutWaitStop(crt_pipe, crt_buf1, 100);
```

```
}

void crt_task_main(BOOLEAN msg_out)
{
    u8 *mp;
    if(msg_out)
        tog = (tog == TRUE ? FALSE : TRUE);
    mp = (tog == TRUE ? crt_buf1 : crt_buf2);
    smx_PipePutWaitStop(crt_pipe, mp, 100);
}
```

In this example, crt_task is a one-shot task. Note that crt_task_init() performs the initial pipe put wait stop operation. Prior to this, the task entry point is changed to crt_task_main(), which expects a msg_out TRUE/FALSE parameter from the put operation. When the put operation in crt_task_init() puts a packet, it starts crt_task at crt_task_main(), where output toggles between buffers unless a buffer does not get put into crt_pipe, in which case, it is put again. The put operation waits up to 100 ticks to put a buffer.

smx_PipeResume

BOOLEAN smx_PipeResume (PICB_PTR pipe)

Type SSR

Summary Resumes tasks waiting on pipe, if wait condition true.

Parameters pipe Pipe handle. Operation is aborted if not valid.

Returns
TRUE Operation performed.
FALSE Operation not performed because of invalid pipe.

Errors SMXE_INV_PICB

Descr Resumes tasks waiting in pipe's task queue. Processes tasks in priority order. For each 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.

If there is no task waiting, then smx_PipeResume() does nothing. If more than one task is waiting on pipe get, each will get one packet and be resumed in FIFO order, for as many packets as are available. Conversely, if more than one task is waiting on pipe put, each will put its packet and be resumed in FIFO order, for as many slots as are available.

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. Intended for use from tasks and LSRs. Is protected from interrupts.

Notes

1. All tasks waiting on a pipe must be waiting for the same thing — to put or to get a packet.
2. A packet pipe (i.e. width > 1) is considered empty unless a full packet is present.

Example

```
void key_LSR(void)
{
    smx_PipeResume(key_pipe);
}

void key_ISR(u8 key_port)
{
    u8 ch;

    ch = input_key(key_port);
    smx_PipePut8(key_pipe, ch);
    smx_LSR_INVOKE(key_LSR, 0);
}
```

In this example, key_ISR is loading key_pipe, a character at a time, then invoking key_LSR to wake up the task, which is waiting to process keys. This is done by calling smx_PipeResume(). Note that key_LSR does not need to know what task is waiting, if any.

smx_PipeStatus

u32 smx_PipeStatus (PICB_PTR pipe, PSS *ppss)

Type SSR

Summary Returns the number of packets in pipe and pipe status information.

Parameters

pipe	Pipe handle. Operation is aborted if not valid.
ppss	Pointer to a pipe status structure supplied by the user. NULL, if no status is desired.

Returns

N	Number of packets in pipe.
0	No packets in pipe or invalid pipe.

Errors SMXE_INV_PICB

Descr Loads width, size, flags, and number of tasks waiting into pipe status structure, if specified, and returns number of packets in pipe. Protected from interrupts.

Note A partial packet is not counted.

Example 1

```
TCB_PTR pipe_input_task;

void regulate_pipe(PICB_PTR pipe)
{
    if(smx_PipeStatus(pipe, 0) > 3)
        pipe_input_task->pri++; /* increase task priority */
    if(smx_PipeStatus(pipe, 0) < 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 and to 2 to decrease it. Note that no Pipe Status Structure is specified, since other status is not of interest.

Example 2

```
void send_msg(const char *);

void increase_msgs(PICB_PTR pipe)
{
    PSS pipe_stat;
    smx_PipeStatus(pipe, &pipe_stat);
    if((pipe_stat.numtasks > 1) && !(pipe_stat.flags & SMX_FL_PUT))
        send_msg("Increase message input rate");
}
```

In this example, if more than one task is waiting for packets, a message is sent to the operator to increase the message input rate. In this case, a pipe status structure (PSS) is specified to determine how many tasks are waiting and if they are waiting to get packets.

smx_Sem

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 Error.

Errors SMXE_INV_SCB

Descr Resumes all tasks waiting at sem with FALSE return values and deactivates their timeouts. Then resets the semaphore count to its original value, when created. This call would normally be used in a recovery situation, such as following a SIG_CTR_OVFL error.

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

Example

```
SCB_PTR printer_avail;

smx_SemClear(&printer_avail);
```

smx_SemCreate

SCB_PTR smx_SemCreate (SMX_SEM_MODE mode, u8 lim, const char *name)

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 for none.

Returns

handle	Semaphore created.
NULL	Insufficient resources or error.

Errors

SMXE_INV_PAR	mode or lim not in range
SMXE_OUT_OF_QCBS	

Descr

Gets a semaphore control block (SCB) from the QCB pool and loads the cbtype, mode, count, lim, and name fields. Returns the address of SCB as the semaphore handle.

An smx 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 UG Semaphores.

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, simple_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");
    simple_sem = smx_SemCreate(SMX_SEM_EVENT, 0, "simple_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 could regulate a processing task which requires four sets of data before starting. simple_sem is a multiple event semaphore. It stores every event received. binary_sem is a binary event semaphore. All events, after the first are ignored.

smx_SemDelete

BOOLEAN smx_SemDelete (SCB_PTR *sem)

Type SSR

Summary Deletes a semaphore.

Compl smx_SemCreate()

Parameters *sem Address of the handle of the semaphore to delete.

Returns

TRUE	Semaphore deleted.
FALSE	Error.

Errors SMXE_INV_SCB

Descr Deletes a semaphore created by smx_SemCreate(). First resumes waiting tasks, giving them FALSE return values and deactivating their timeouts. Then clears the semaphore control block, releases it to the QCB pool, and clears sem.

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

Example

```
SCB_PTR printer_avail;

smx_SemDelete(&printer_avail);
```

smx_SemPeek

u32 smx_SemPeek (SCB_PTR sem, SMX_PK_PARM par)

Type SSR

Summary Returns the current value for the parameter specified.

Parameters

sem	semaphore to peek.
par	Argument to return.

Returns

value	Value of par.
0	Value, unless error.

Errors

SMXE_INV_SCB	Invalid message exchange handle.
SMXE_INV_PARM	Invalid argument.

Notes 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	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_SemSignal

BOOLEAN smx_SemSignal (SCB_PTR sem)

Type SSR

Summary Signals a semaphore. Operation depends upon the semaphore mode — see below.

Compl smx_SemTest(), smx_SemTestStop()

Parameters sem Semaphore to signal.

Returns

TRUE	Signal sent.
FALSE	Error.

Errors SMXE_INV_SCB
SMXE_SIG_CTR_OVFL

Descr

mode **action**
RSRC: Resume top task with TRUE, else for count < lim, count++.
EVENT: Resume top task with TRUE, else:
  if lim == 1 and count < lim, count = 1.
  if lim != 1 and count < 255, count++, else: SMXE_SEM_CTR_OVFL.
THRES : If count < 255: count++ else: SMXE_SEM_CTR_OVFL
  If task waiting and count >= lim: count - lim and resume top task with TRUE.
GATE : Resumes all waiting tasks with TRUE.

RSRC and EVENT semaphores with lim == 1 are called binary semaphores. For more discussion concerning usage see UG Semaphores.

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

Notes If the mode is not recognized, SMXE_INV_SCB is reported.

Example

```
SCB_PTR  ready_for_msg;
TCB_PTR  calc;
XCB_PTR  input;

 /* create a binary resource semaphore */
ready_for_msg = smx_SemCreate(RSRC, 1, "ready_for_msg");

void calc_main(void)
{
    u8 *dp;
    MCB_PTR msg;

    while (1)
    {
        smx_SemSignal(ready_for_msg);
        if(msg = smx_MsgReceive(input, &dp, SEC))
            /* process block */
        else
            /* report problem */
    }
}
```

In this example, calc signals the ready_for_msg semaphore then waits at the input exchange for a message. Some other task should be waiting at the ready_for_msg semaphore, ready to send another message. If a message is received in less than a second, calc processes it. Otherwise, calc reports that there is a problem.

smx_SemTest

BOOLEAN smx_SemTest (SCB_PTR sem, u32 timeout)

Type SSR

Summary Tests if sem has a pass condition. If so, decreases sem count and continues ct. Otherwise, suspends ct on sem.

Compl smx_SemSignal()

Parameters

sem	Semaphore to test.
timeout	Timeout in ticks.

Returns

TRUE	OK.
FALSE	Error or timeout.

Errors SMXE_INV_SCB

SMXE_WAIT_NOT_ALLOWED

Descr

mode	**action**
RSRC:	If count > 0: decrement count and continue current task with TRUE, else priority enqueue it in sem wait queue and activate its timeout, unless NO_WAIT or INF. If NO_WAIT, continue current task with FALSE.
EVENT:	Same.
GATE:	Same.
THRES :	If count >= lim: count = count - lim and continue current task with TRUE, else priority enqueue it in sem wait queue and activate its timeout, unless NO_WAIT or INF. If NO_WAIT, continue current task with FALSE.

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 (i.e. timeout must be NO_WAIT).

Notes (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", ...);

start_cycle = smx_SemCreate(GATE, 1, "start_cycle"); /* gate semaphore */
data_ready = smx_SemCreate(THRES, N, "data_ready"); /* threshold semaphore */

void init_main(void)
{
    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(void)
{
    while(1)
    {
        if (smx_SemTest(start_cycle, TMO))
        {
            /* acquire data and store in global area */
            smx_SemSignal(data_ready);
        }
        else
            /* notify of timeout or error */
```

```
    }
}

void process_main(void)
{
    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 get tasks wait at the start_cycle gate semaphore. When all are waiting, a signal is given to start them all at once. As each get task finishes, it signals the data_ready threshold semaphore, then goes back to wait at the start_cycle gate semaphore. After N signals, the process task is resumed. It processes the data, then signals the start_cycle gate semaphore and the operation repeats. The above example also illustrates how to handle timeouts or errors for smx_SemTest(). If the get tasks need to wait on data or resources, then multiple get tasks will result in more efficient usage of the processor than one get task, since some get tasks can run while others are waiting.

Example 2

```
SCB_PTR  printer_ready; /* binary resource semaphore */
TCB_PTR  t2a, t3a;

printer_ready = smx_SemCreate(RSRC, 1, "printer_ready");

void t2a_main(void)
{
    ...
    smx_SemTest(printer_ready, TMO);
    /* send data to printer */
    smx_SemSignal(printer_ready);
}

void t3a_main(void)
{
    ...
    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 use this semaphore, as shown, in order to avoid conflicts.

smx_SemTestStop

void smx_SemTestStop (SCB_PTR sem, u32 timeout)

Type limited SSR — task only

Summary Operates the same as smx_SemTest(), except that ct is always stopped, then restarted when it is time for it to run.

Compl smx_SemSignal()

Parameters

sem	Semaphore to test.
timeout	Timeout in ticks.

Errors SMXE_OP_NOT_ALLOWED
SMXE_INV_SCB

Descr See smx_SemTest() for operational description. ct always stops, 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) smx_lockctr is cleared if called from a task.

TaskMain void task_main(BOOLEAN par)

par

TRUE	Got semaphore.
FALSE	Error or timeout.

Example

```
SCB_PTR data_ready, start_cycle;

for (i = 0; i < N; i++)
{
    get[n] = smx_TaskCreate(get_main, PR2, 0, 0, name[n]);
    smx_TaskStartPar(get[n], 1);
}

void get_main(BOOLEAN pass)
{
    if (pass)
    {
        /* acquire data and store in global area */
        smx_SemSignal(data_ready);
        smx_SemTestStop(start_cycle);
    }
    else
        /* notify of timeout or error */
}
```

This is equivalent to the first example 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 par == 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 uses a stack.

As noted in smx_SemTest() Example 1, the reason for using multiple get tasks is that they can separately wait for inputs, thus improving processor efficiency. As structured above, a get task must retain its stack while waiting for an input, hence the stack pool might need, for example, to have 5 stacks to efficiently support 10 get tasks. The number of stacks could be reduced by redesigning the get tasks to stop when waiting for inputs.

smx_SSR

smx_SSR_ENTER

void smx_SSR_ENTERn (u32 id, u32 par1, ... , u32 parn)

Type macro

Summary Used to begin a system service routine (SSR).

Compl smx_SSR_EXIT()

Parameters

id	SSR ID — see xdef.h
par1-6	Parameters of the call.

Returns none

Descr All system service routines (SSRs) must begin with smx_SSR_ENTERn(), which first increments smx_srnest, then calls the corresponding smx_EVBLogSSRn() function, if SMX_CFG_EVB in xcfg.h is set. id identifies the SSR. It is put first in the event buffer (EVB) record after the beginning of record marker (0x5555rrss, where rr = record type and ss = its size). The SSR's parameters are placed next in the EVB record, in order. Seven different enter macros and log functions provide an exact match of SSR parameters to record parameters, so that all are stored without wasted space.

It is possible to selectively enable logging of groups of SSRs. This helps to save space in the EVB, by eliminating unneeded SSR records. In the SSR ID, the SSR Group field allows SSRs to be grouped into 9 groups: SG0, which is never logged, and SG1-8, which are logged if the corresponding bit of smx_evben is set. All smx SSRs are initially put into SG1. Selected SSRs can be moved into other groups, by changing their SG field (in their IDs in xdef.h) so that groups of SSRs can be selectively enabled or disabled. An SSR can belong to more than one group (since groups are specified by bits).

The flags in smx_evben can be controlled during debugging via smxAware. When EVB is initialized from smx_Go() all flags are set.

Custom SSRs require new IDs, if they are to be logged. IDs are defined in xdef.h. Each ID specifies: module (01 for smx), SSR Group, number of parameters, and function ID. The ID format is 0xMMSSPIII. For custom SSRs, it is probably best to assign a new module number so they can be kept separate from new smx SSRs in future releases. We plan is to assign IDs to other SMX products, in the future, so it would be best to pick a high number.

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. Note that all intermediate returns and error exits must also call smx_SSR_EXIT(), as shown.

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.

When creating a custom SSR, it is best to start with an smx SSR that is close to what you want and modify it. For more information see UG, Service Routines, custom SSRs.

smx_SSR_EXIT

u32 smx_SSR_EXIT (ret, id)

Type function

Summary Used to end a system service routine (SSR).

Compl smx_SSR_ENTER()

Parameters

ret	Value to return (or preliminary value for calls that wait).
id	SSR ID.

Returns result:

1. SSR calls which can be immediately satisfied (e.g. smx_MsgReceive() and a message is waiting at the exchange): The same value specified in the value parameter is returned to the caller (or passed in as the start parameter — see note 3a).

2. SSR calls which must wait to be satisfied (e.g. smx_MsgReceive() and no message is waiting at the exchange): In this case, the current task will be suspended or stopped. tcb.rv must be preloaded with the value that is appropriate if a timeout occurs (normally FALSE or 0).
   a. If the waiting condition is satisfied before timeout elapses (e.g. a message is sent to the exchange), tcb.rv is overwritten with the appropriate return value by the complementary call (in this case, smx_MsgSend() sets tcb.rv to the message handle). tcb.rv is what is eventually returned to a task that resumes or passed to a stopped task that restarts.

b. If the waiting condition is not satisfied before the timeout elapses, the preloaded value is returned or passed.

3. Return method
   a. suspend SSRs and others: return value in the appropriate register, as is done for any C function.
   b. stop SSRs: “return” value by passing it as the parameter to the task when it is restarted.

Descr For a suspend call, if smx_srnest > 1, returns to the point of interrupt with the SSR result. For a stop call, restarts the current task and passes the SSR result as the main function parameter. If smx_srnest == 1, tests if there is an LSR in lq or if a scheduler flag (STOP, SUSP, or TEST) is set. If so, branches to the prescheduler, which calls the LSR scheduler or the task scheduler, as appropriate. In this case, control may pass to another task and the current task may be suspended or stopped. If there is no waiting LSR an no scheduler flag set, operation is the same as for smx_srnest > 1.

All SSRs must end with:

```
return(smx_SSR_EXIT(value, id));
```

or

```
smx_SSR_EXIT(0, id);
```

if there is no return value.

Example See example above.

smx_Sys

smx_SysEtimeGet

u32 smx_SysEtimeGet (void)

Type Bare macro

Summary Gets the current elapsed time in ticks.

Parameters none

Returns smx_etime

Descr Returns etime in ticks. See UG Timing, etime and stime.

Example

```
u32 etime;

etime = smx_SysEtimeGet();
```

smx_SysPseudoHandleCreate

void* smx_SysPseudoHandleCreate (void)

Type Bare function

Summary Creates a pseudo handle to identify an objects 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, LSRs, and user-define events. These can be used in the smx_EVB_LOG macros to log when ISRs and LSRs run, in the Event Buffer. A pseudo handle is a void pointer that is in the range of SMX_PSEUDO_HANDLE_MIN to SMX_PSEUDO_HANDLE_MAX, which are defined in xdef.h. Each new pseudo handle is 1 greater than the previous one created.

Note Requires SMX_CFG_EVB.

Example See smx_EVB_LOG().

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. See UG Timing, etime and stime.

Example

```
u32 stime;

stime = smx_SysStimeGet();
```

smx_SysPowerDown

BOOLEAN smx_SysPowerDown (u32 sleep_mode)

Type SSR

Summary Puts processor into specified sleep mode. Restores all tick-related timing when power resumes.

Parameters sleep_mode Sleep mode desired (e.g. DEEP_SLEEP).

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 BSP or Base 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 tq, the first task in smx_TicksEQ(), 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 processes it. This process continues until all events, which would have occurred during power down, have been processed. 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 lq is large enough to handle all LSR invocations. If not, earlier LSRs will be lost.

After tick recovery is complete, stime is updated and the SSR is exited. Following this, LSRs then tasks will execute in the order invoked or resumed. 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(void)
{
    while(TRUE)
  {
        ...
        if (idle_done)
            smx_SysPowerDown(SLEEP);
    }
}
```

This is the normal use of smx_SysPowerDown() — i.e. at the end of an idle loop, after idle 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 *hdl)

Type SSR

Summary Returns control block type for handle.

Parameters hdl Handle.

Returns type Type of control block.
0 Control block type is not recognized.

Descr Returns the control block type of the control block pointed to by hdl. Returns NULL if hdl does not point to a valid control block. hdl 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

smx_TaskBump

BOOLEAN smx_TaskBump (TCB_PTR task, u8 pri)

Type SSR

Summary Changes task priority 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	
SMXE_INV_PRI	pri > SMX_MAX_PRI
SMXE_INV_TCB	

Descr Changes task priority to pri, unless pri == SMX_PRI_NOCHG: task->normpri = pri and task->pri = pri, if task owns no mutexes. Otherwise task->pri is promoted, but not demoted. Whether or not task->pri is changed, task is requeued at the end of its priority level, unless it is in a FIFO queue or not in a queue. 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 the current task is not locked, it may be preempted.

The current task can bump itself. This can result in it being preempted.

Example

```
void taskA_main(void)
{
    smx_TaskUnlock();
    while (smx_TaskBump(smx_ct, SMX_PRI_NOCHG))
    {
        /* do main function */
    }
}
```

After the first bump, each time taskA completes its main function, it bumps itself to the end of its priority level. This allows other tasks at the same priority level to run. If they also bump themselves to the end, this is good way to implement cooperative multitasking.

smx_TaskCreate

TCB_PTR smx_TaskCreate (FUN_PTR fun, u8 pri, u32 stack_size, u32 flags, const char *name)

Type	SSR	
Summary	Creates a task with fun as its main function and with the parameters specified.	
Compl	smx_TaskDelete()	
Parameters	fun	Main function.
	pri	Priority.
	stack_size	Stack size.
	flags	Flags to specify characteristics of the task: SMX_FL_LOCK start locked SMX_FL_NONE no flags specified to set
	name	Name to give task or NULL for none.
Returns	handle	Task created.
	NULL	Insufficient resources or error.
Errors	SMXE_INV_PRI	pri > SMX_MAX_PRI.
	SMXE_INV_DAR	Insufficient DAR to create stack & TCB pools on first call.
	SMXE_INSUFF_HEAP	Insufficient heap for permanent stack.
	SMXE_OUT_OF_TCBS	TCB pool 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.

If stack_size is 0, the task will be given a stack from the stack pool when it begins running. That stack is not permanently bound to the task and will be released when the task next stops. Otherwise, a permanent stack is assigned to the task from the heap of stack_size bytes. This stack is permanently bound to the task. The stack bottom is aligned as specified by SB_STACK_ALIGN. As a consequence, the actual stack size may be less than expected. For more details see UG Stacks.

SMX_FL_LOCK, sets the task's start locked flag. This causes the task to always start in the locked state, which is useful for initialization and for one-shot tasks; it is similar to an ISR starting with interrupts disabled. 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. Short names are recommended. HT is used by smxAware.

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 location named for the task.

Example

```
TCB_PTR taskA, taskB;

void taskX_main(void)
{
    smx_TaskLock();
    taskA = smx_TaskCreate(taskA_main, PRI_NORM, 0, SMX_FL_NONE, "taskA");
    smx_TaskStart(taskA);
    taskB = smx_TaskCreate(taskB_main, PRI_LO, 1000, SMX_FL_NONE, "taskB");
    smx_TaskStart(taskB);
    smx_TaskStart(taskA);
}
```

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. Note that taskX is automatically unlocked when it auto stops.

smx_TaskDelete

BOOLEAN smx_TaskDelete (TCB_PTR *task)

Type SSR

Summary Releases resources owned by task and deletes it.

Compl smx_TaskCreate()

Parameters task — Task to delete.

Returns
TRUE — Task deleted.
FALSE — Error.

Errors
SMXE_INV_PARM — smx_ct was passed
SMXE_INV_TCB

Descr Releases all owned blocks, messages, timers, and mutexes. Dequeues task if it is in a queue. Frees the task’s stack back to the heap, if permanent, or to the stack pool, if not. Deactivates the task timeout, releases its TCB back to the TCB pool, then clears its handle. At this point, task has ceased to exist. An attempt to use task will result in an SMXE_INV_TCB error.
If task has already been deleted (task == NULL), operation is aborted and SMXE_INV_TCB is reported.

The current task may delete itself. However, its handle must be passed in, not smx_ct (which is an alias pointer). Passing smx_ct is not permitted because it would be cleared, causing smx to malfunction.

Note Heap are not automatically released because they may contain blocks or messages owned by other tasks or LSRs. Hence, avoiding heap memory leaks is left to application code.

Example

```
TCB_PTR old_task;

   if (!smx_TaskDelete(&old_task))
      /* check old_task handle */
```

smx_TaskHook

BOOLEAN smx_TaskHook (TCB_PTR task, FUN_PTR entry, FUN_PTR exit)

Type SSR

Summary Hooks entry and exit routines to task, which run when the task is resumed or suspended

Compl smx_TaskUnhook()

Parameters

task	Task to hook entry and/or exit routines to.
entry	Entry routine address.
exit	Exit routine address.

Returns

TRUE	Task hooked.
FALSE	Error.

Errors SMXE_INV_TCB

Descr smx_TaskHook() is used to hook entry and exit routines to task. It hooks the specified task by setting task->flags.hookd. Then the address of entry is loaded into task->hookentry and exit is loaded into task->hookexit. If only one routine is needed, pass NULL for the other. This causes smx smx_NullF() to be hooked, which simply returns. The hookd flag is not set if both entry and exit are NULL.

Hooking operates as follows: when task is to be resumed, the scheduler calls the entry routine. When task is to be suspended, the scheduler calls the exit routine. Exit and entry routines can be used to transparently preserve extended task states on a task-specific basis. For example, coprocessor registers can be saved and restored only for tasks using the coprocessor.

Exit and entry routines are normal C functions, defined as follows:

```
void routine(void)
```

To minimize impact upon interrupt latency, they are called with interrupts enabled. Interrupts can be disabled, if necessary to prevent ISRs from running. Exit and entry routines are task-safe, so there is no need to be concerned about other tasks, LSRs, or SSRs running. SSRs must not be called from these routines.

Exit and entry routines operate as extensions to the scheduler, hence their execution times directly add to task suspend and resume times. Therefore, they should be as fast as possible and may need to be written in assembly language. If so, be sure to preserve non-volatile registers, as with any C function.

Notes

1. Stopping a task does not call the exit routine and starting a task does not call the entry routine. Hence, in one-shot tasks it may be necessary to call these functions directly if it is expected that an extended state will be preserved across a stop/start transition.

2. Due to the fly-back mechanism of the smx scheduler, it is possible for entry and exit to be called without the task actually running in between.

Example 1

```
void atask_main(void)
{
    smx_TaskHook(smx_ct, atask_entry, atask_exit);
    /* other initialization */
    while(work_to_do)
    {
        /* task functions performed */
    }
    smx_TaskUnhook(smx_ct);
}

PUBLIC _atask_entry, _atask_exit

_atask_entry:
    ; restore variables, stacks, etc. used by atask
    ret

_atask_exit:
    ; save variables, stacks, etc. used by atask
    ret
```

In this example, atask hooks the atask_entry and atask_exit routines to itself. While atask loops, it can be preempted, suspended, or resumed. If atask leaves the while loop, it unhooks itself, which is not essential — rehooking the exit and entry routines, when it restarts, will cause no harm, but might confuse someone reading the code.

Example 2

```
smx_TaskHook(smx_ct, atask_entry, NULL);
```

Here, the exit routine is not necessary — perhaps the atask_entry just sends a pulse.

smx_TaskLocate

VOID_PTR 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
SMXE_INV_TCB

Descr Returns a pointer to the queue that task is in, if it is in a queue, or NULL if not in a queue. Aborts and reports SMXE_BROKEN_Q if no queue control block is found.

Example

```
BOOLEAN resume_task(TCB_PTR task, MCB_PTR ack_msg)
{
    pass = 0;
    CB_PTR  q;

    q = (CB_PTR)smx_TaskLocate(task);
    switch (q->cbtype)
    {
        case SMX_CB_XCHG:
            smx_MsgSendPR(ack_msg, q, 0, 0);
            pass = TRUE;
            break;
        case SMX_CB_SEM:
            smx_SemSignal(q);
            pass = TRUE;
    }
    return(pass);
}
```

This function allows resuming a task which may be waiting at an exchange for an ack_msg or for a signal at an exchange. 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

Descr Increments smx_lockctr, unless smx_lockctr would reach SMX_CFG_LOCK_NEST_LIMIT, in which case, it aborts and reports excess locks. 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 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(void)
{
    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

Descr If smx_lockctr is not 1, gives a warning. Clears smx_lockctr and tests for preemption. Called from smx_TaskUnlock() when smx_lockctr == 1. Also recommended to be called, instead of smx_TaskUnlock(), at the end of lock nesting in the task main function, as a precaution to assure that the lock counter is zero.

Example

```
u32 hour;

void hourly_main(void)
{
    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 assures that the task will be unlocked even if there have been more locks than unlocks.

smx_TaskPeek

u32 smx_TaskPeek (TCB_PTR task, SMX_PK_PARM par)

Type SSR

Summary Returns the current value for the parameter specified.

Parameters

task	Task to peek at.
par	Parameter to return value.

Returns	value	Value of par.
	0	Value, unless error.

Errors SMXE_INV_TCB
SMXE_INV_PARM

Descr This service allows peeking at a task. Valid arguments are:

SMX_PK_NEXT	Next task linked to task in a queue; NULL, if none.
SMX_PK_LAST	Previous task linked to task in a queue; NULL, in none.
SMX_PK_STATE	State.
SMX_PK_ERROR	Error last reported for task.
SMX_PK_INDEX	Index of task in TCB pool. Also used for timeout array.
SMX_PK_NAME	Name.
SMX_PK_PRI	Priority.
SMX_PK_PRINORM	Normal priority.
SMX_PK_TMO	Timeout remaining.

Notes

1. SMX_PK_ERROR cannot be used for the current task because it will return the result for itself (i.e. SMXE_OK). Instead, use SMX_ERR.

Example

```
TCB_PTR  atask;
u32 time;

time = smx_TaskPeek(atask, SMX_PK_TMO);
if (time > 10)
      smx_TaskResume(atask);
```

In this example, if atask has more than 10 ticks left to wait, resume it with a failure indication.

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	Error.

Errors SMXE_INV_TCB

SMXE_BROKEN_Q

Descr Dequeues task from any queue it may be in. If in an event queue the differential count of the following task is increased by the differential count of task. If task is in a pipe queue, performs the equivalent of smx_PipeResume(). Then if task is still not resumed, resumes it with a FALSE return for the put or get operation. For any other queue, task is resumed as if a timeout had occurred. If the queue is broken, operation is aborted and an error reported.

smx_TaskResume() can cause a task to be removed from a queue it was just put into (e.g. a semaphore). The call it had suspended on (e.g. smx_SemTest()) will then appear to fail and task's timeout is disabled.

Operation is the same for bound and unbound tasks. Hence smx_TaskResume() can be used when it is not known which mode task is in. For example, if task had been stopped by smx_SemTestStop() it will be restarted, but if it had been suspended by smx_SemTest(), it will be resumed.

If the current task is unlocked, it will be preempted, if task has higher priority. The current task may resume itself. The net result of this is that it is moved to the end of its rq level. If the current task is still the top task in rq or if it is locked, it is continued. Otherwise, it is preempted by the new first task in the level.

Example

```
void taskn_main(void)
{
    do
    {
        /* perform operations for this task */
    } while(smx_TaskResume(smx_ct));
}
```

This is an example of round-robin scheduling. If other equal priority tasks are written this way and are in rq, each will run, then move itself to the end of the rq level by calling smx_TaskResume(smx_ct). When all of these tasks have run, the process will repeat. Higher priority tasks can preempt the round-robin tasks, but lower priority tasks are locked out. For this reason, round-robin scheduling is normally used only at priority 0.

smx_TaskSetStackCheck

BOOLEAN smx_TaskSetStackCheck (TCB_PTR task , BOOLEAN state)

Type SSR

Summary Enables stack checking if state is ON; otherwise, disables it.

Parameters

task	Task whose stk_chk flag should be changed.
state	TRUE to set; FALSE to clear.

Returns

TRUE	Flag changed
FALSE	Error

Errors SMXE_INV_TCB

Descr If task->flags.stk_chk is 1, the task's stack is checked for overflow when it is suspended, stopped, or deleted. If task->flags.stk_chk is 0, these tests are not performed. This service is used when a foreign stack is in use (such as when a BIOS call is made) in order to avoid false stack overflow errors.

Example

```
void function(void)
{
    /* stack checking is initially on */
    ...
    smx_TaskSetStackCheck(smx_ct, FALSE)
    /* call function which changes stacks */
    smx_TaskSetStackCheck(smx_ct, TRUE)
    ...
}
```

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.

smx_TaskSleep

BOOLEAN smx_TaskSleep (u32 time)

Type limited SSR — tasks only

Summary Suspends the current task until the specified system time (time == stime).

Parameters time Time to awaken, in seconds.

Returns

TRUE	ct has been delayed.
FALSE	No delay due to invalid time.

Errors

SMXE_INV_TIME	time <= stime
SMXE_OP_NOT_ALLOWED	Called from an LSR

Descr If time is greater than stime, the current task is suspended, and its timeout is set to

smx_etime + (time - smx_stime) * smx_cf.sec

where smx_cf.sec is the number of ticks per second. The amount added to etime must be less than 2exp31. 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.

If time is less than or equal to stime, continues the current task, returns FALSE, and reports SMXE_INV_TIME.

Notes (1) If called from an LSR, aborts operation and returns to LSR.
(2) smx_lockctr is cleared if called from a task.

Example 1

```
smx_TaskSleep(smx_SysStimeGet() + 10);    /* sleep until 10 seconds from now */
```

Example 2

```
u32 next_hour = smx_stime + (3600 - (smx_stime % 3600));
smx_TaskStart(hourly);

void hourly_main(void)
{
    while(smx_TaskSleep(next_hour))
    {
        /* perform hourly function */
        next_hour += (3600 - (next_hour % 3600));
    }
}
```

In this example, the hourly task wakes up at the start of the next hour and performs it 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. Resolution is one second.
Parameters	time Time to sleep, in seconds from now.
Errors	SMXE_INV_TIME time <= stime SMXE_OP_NOT_ALLOWED Called from an LSR
Descr	See smx_TaskSleep() for operational description. ct always stops, 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) smx_lockctr is cleared if called from a task.
TaskMain	void task_main(BOOLEAN par)
par	TRUE Current task has been delayed FALSE No delay due to invalid time

Example

```
TCB_PTR  hour;
u32  next_hour;
#define ENTER -1;

next_hour = smx_stime;
smx_TaskStartPar(hour, FALSE);

void hourMain(BOOLEAN pass)
{
    if (pass)
        /* perform hourly function */
    next_hour += (3600 - (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 hour 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)
BOOLEAN smx_TaskStartPar (TCB_PTR task, u32 par)
BOOLEAN smx_TaskStartNew (TCB_PTR task, u32 par, u8 pri, FUN_PTR fun)

Type SSR

Summary smx_TaskStart() puts task into the ready queue at the end of its priority level. Stops it first, if necessary. smx_TaskStartPar() starts task and passes par to it. smx_TaskStartNew() restarts a task with fun as its main function and pri as its priority, and passes par.

Compl smx_TaskStop()

Parameters

task	Task to start.
par	Parameter to pass to task.
fun	New main function.
pri	New priority.

Returns

TRUE	Task started.
FALSE	Error.

Errors

SMXE_BROKEN_Q
SMXE_INV_PRI
SMXE_INV_TCB

Descr All three 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 in a pipe queue, calls smx_PipeResume_F(); if task is not restarted by this, restarts it with a FALSE parameter for the put or get operation. Task's stack pointer (tcb.sp) is cleared, its timeout is stopped, and its stack is released back to the stack pool, unless permanent. (If SMX_CFG_STACK SCAN is TRUE, the stack is actually put into the scanstack pool and later moved to the freestack pool after it has been scanned and refilled with the test pattern.)

If the current task starts itself, the result is that it is stopped and moved to the end of its rq level. Its stack is later released by the scheduler. These actions occur even if the current task is locked. If the current task is still the top task in 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 ct.

When task is restarted, it is given a new stack, if the old one was released, and it starts from the beginning of its main function. 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_TaskStartPar() is a convenient way to start a task and pass a parameter to it, if its main function accepts a parameter. If not, the parameter is ignored. StartPar() is used primarily to start one-shot tasks. If task is stopped on a pipe, TRUE or FALSE from the stopped pipe operation will be overwritten with par.

smx_TaskStartNew() allows changing the task's main function, its priority, and passing a parameter if its main function accepts a parameter. smx_TaskStartNew() loads fun into task->fun; loads pri into task->priority and task->normpri, unless pri == SMX_PRI_NOCHG; resets task->flags.hookd flag; and starts task.

TaskMain void task_main(u32 par) or void task_main(void)

par is passed via task->rv for smx_TaskStartPar() or smx_TaskStartNew(). task->rv is unaffected for smx_TaskStart(). If the processor has separate data and address registers, see the note concerning task main function parameters in smx_Calls Notes and Restrictions.

Example 1

```
TCB_PTR load_task, waiting_task;
XCB_PTR data_msgs;

waiting_task = smx_TaskCreate((FUN_PTR)waiting_task_main, PRI_HI, 0, SMX_FL_NONE,
                                                  "waiting_task");


void load_task_main(void)
{
    u8 *dp;
    MCB_PTR msg;

    while (msg = smx_MsgReceive(data_msgs, &dp, TMO))
    {
        /* load msg */
        msg->onr = waiting_task;
        smx_TaskStartPar(waiting_task, (u32)msg);
    }
}


void waiting_task_main(MCB_PTR msg)
{
    /* process msg */
}
```

load_task obtains a message, changes its owner to waiting_task and starts waiting task, with msg as its parameter. It is assumed that waiting_task has equal or greater priority so it will run and process the message before load_task gets the next message. Note that it is a one-shot task.

Example 2

```
void tx_doneLSR(void)
{
   smx_TaskStartPar(tx, 0);
}

void tx_main(u32 timeout)
{
   if(timeout)
      /* resend message */
   else
      /* send next message */
      smx_EventQueueCountStop(smx_TicksEQ, TX_TIMEOUT, SMX_TMO_INF);
}
```

tx_doneLSR 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 3

```
void appl_init(void)
{
   gp = smx_TaskCreate(gp_init, PRI_MAX, 0, SMX_FL_NONE, "gp");
   smx_TaskStart(gp);
}

void gp_init(void)
{
   /* perform initialization */
   smx_TaskStartNew(smx_ct, 0, PRI_NORM, gp_run);
}

void gp_run(void)
{
   /* perform normal operations */
}
```

In this example, the gp task is initially started at maximum priority with gp_init() as its code. When initialization of gp is complete, smx_TaskStartNew() causes gp to start gp_run() with normal priority. This is commonly used for one-shot tasks, which require initialization.

smx_TaskStop

BOOLEAN smx_TaskStop (TCB_PTR task, u32 timeout)

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.

Returns TRUE OK (not possible if task is stopping itself).
FALSE Error.

Errors SMXE_INV_TCB

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. If its stack is from the stack pool, it is put into the freestack pool, unless SMX_CFG_STACK_SCAN is true. If so, the stack is put into the scanstack pool so that it will be scanned and cleared by smx_StackScan(), then released to the freestack pool. task's stack pointer (tcb.sp) is cleared, so all local variables and task's run context are lost. task's timeout is set according to the value of timeout (see Notes below). When the timeout elapses or if it was 0, task is restarted — i.e. it is put into rq at the end of its priority level.

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 after a delay. smx_lockctr() is cleared due to smx_TaskStop() or any other stop function.

A task may also stop itself, even if it is locked. If a task does stop itself, smx_TaskStop() is the last statement it executes. The current task may also be stopped by an LSR, even if locked.

Notes (1) task's timeout is set to timeout[tn] = smx_etime + timeout, unless timeout == SMX_TMO_NOWAIT, SMX_TMO_NOCHG, or SMX_TMO_INF. In the first and third cases, timeout[tn] is made inactive. In the second case, timeout[tn] is not changed. In the second and third cases, task is not immediately enqueued in rq.
(2) smx_lockctr is cleared if task is ct.

TaskMain void task_main(u32 par) or void task_main(void)

task->rv is unaffected. If the processor has separate data and address registers, see the note concerning task main function parameters in smx_Calls Notes and Restrictions.

Example

```
TCB_PTR task_stop, t;

/* t needs to be stopped */
smx_TaskStartPar(task_stop, (u32)t);


void task_stop_main(TCB_PTR t)
{
    /* release all blocks, msgs, mutexes, and heap blocks owned by t */
    smx_TaskStop((TCB_PTR)t, SMX_TMO_INF);
}
```

task_stop releases all objects that t owns, then stops it indefinitely, so that it can cause no further damage. t ends up in a dormant state from which it can be restarted only by another task.

Task Autostop

return (par)
or
}

Type C statement

Parameters par Value passed to task if 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.

TaskMain

```
u32  task_main(u32  par)
{
    ...
    return(par);
}
```

OR

```
void  task_main(void)
{
      ...
}
```

Example 1

```
TCB_PTR  comm;
PCB_PTR  blocks;
u8 *bp;

void appl_init(void)
{
      BCB_PTR block;

      comm = smx_TaskCreate(comm_main, PRI_NORM, 0, SMX_FL_NONE, "comm");
      block = smx_BlockGet(blocks, &bp, 0);
      block->onr = comm;
      comm->rv = (u32)bp;
      smx_TaskStart(comm);
}

u32 comm_main(u32 bp)
{
      u8 *dp = (u8*)bp;

      /* use dp as working pointer to access the block */
      return(bp);
}
```

In the above, appl_init() first creates the comm task. It then gets a block from the block pool, transfers it to comm, and starts comm. In the above, comm accepts the block pointer passed to it by appl_init() 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

```
      return((u32)smx_MsgReceive(input, 0, TMO));
}
```

and

```
      smx_MsgReceive(input, 0, TMO);
}
```

and

```
      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 does not use a stack block while waiting for a message whereas the other two do. Hence it is the best way to implement the smx_MsgReceive in this case (since the task eventually stops in all cases).

smx_TaskSuspend

BOOLEAN smx_TaskSuspend (TCB_PTR task, u32 timeout)

Type SSR

Summary Dequeues task and sets its timeout to resume after timeout ticks.

Compl smx_TaskResume()

Parameters task Task to suspend.

Returns TRUE Task suspended.
FALSE Error.

Errors SMXE_INV_TCB

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 the current task is suspending another task, the other task's return value is not changed unless it was in an event queue. If task is already suspended or stopped, this call has no effect, except to possibly change its timeout. task's timeout is set according to the value of timeout (see Notes below).

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 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 ct is locked, smx_lockctr is cleared. Hence, it no longer will be locked when it resumes.

Notes

1. task's timeout is set to timeout[tn] = smx_etime + timeout unless timeout == SMX_TMO_NOWAIT, SMX_TMO_NOCHG, or SMX_TMO_INF. In the first and third cases, timeout[tn] is made inactive. In the second case, timeout[tn] is not changed. In the second and third cases, task is not immediately enqueued in rq. The final stack pointer is loaded into smx_ct->sp, so task can be resumed.
2. Clears smx_lockctr if task is ct. 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 ct to the end of its ready queue level and thus ct may actually be suspended.

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_TaskUnhook

BOOLEAN smx_TaskUnhook (TCB_PTR task)

Type SSR

Summary Unhooks the entry and exit routines from task.

Compl smx_TaskHook()

Parameters task Task to unhook.

Returns

TRUE	Task unhooked.
FALSE	Error.

Errors SMXE_INV_TCB

Descr Clears the hookd flag and the entry and exit routine pointers in task's TCB. Since the exit and entry routines run each time the task is suspended and resumed, task switching time is saved if the hooking is done only for sections that need it, rather than for the whole task.

Another reason to unhook a task is if it is to be restarted and there might be a problem if it starts out hooked, such as if a buffer does not exist at entry to the task since the task itself allocates it.

Example See smx_TaskHook()

smx_TaskUnlock

BOOLEAN smx_TaskUnlock (void)

Type Bare function

Summary Decrements the lock counter. If it becomes 0, unlocks the current task and tests 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 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, ct is preempted.

Note Any smx function that might suspend or stop the current task will also clear the lock, whether or not suspension or stopping actually occurs.

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(void)
{
    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. The lock 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 lock counter. If it becomes 0, unlocks the current task, 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, when 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.

Example

```
u32 hour;

void hourly_main(void)
{
    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_Timer

smx_TimerDup

BOOLEAN smx_TimerDup (TMCB_PTR *tmrbp, TMCB_PTR tmra, const char *name)

Type SSR

Summary Creates a duplicate timer tmrb from tmra and enqueues it after tmra in tq.

Parameters

tmra	Timer to duplicate.
tmrbp	Pointer to location for tmrb handle.
name	Name to give timer or NULL for none.

Returns

TRUE	Timer duplicated.
FALSE	Error. Timer not duplicated.

Errors

SMXE_INV_PARM
SMXE_INV_TMCB
SMXE_OUT_OF_TMCBS
SMXE_TMR_STOPPED

Descr Gets a TMCB 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.

Returns FALSE, and reports SMXE_INV_PARM error, if tmrbp == NULL, *tmrbp != NULL (tmrb is in use), or tmra == NULL (tmra has timed-out or has been stopped). Returns FALSE and reports SMXE_INV_TMCB error, if tmra is not a valid timer handle. Returns FALSE and reports SMXE_OUT_OF_TMCBS error if no TMCB is available for tmrb.

Example

```
TMCB_PTR atmr, btmr;

smx_TimerStart(&atmr, 10, 0, aLSR, "atmr");
smx_TimerDup(&btmr, atmr, "btmr");
```

In this example, btmr is created as a duplicate of atmr and it is enqueued in tq immediately after atmr with 0 differential count. btmr can then be changed, if desired, by any services described below.

smx_TimerPeek

u32 smx_TimerPeek (TMCB_PTR tmr, SMX_PK_PARM par)

Type SSR

Summary Returns the current value for the parameter specified.

Parameters

Parameters	tmr	Timer to peek at.
	par	Parameter to return to return value.

Returns	value	Value of par.
	0	Value, unless error.

Errors SMXE_INV_PARM
SMXE_INV_TMCB
SMXE_TMR_STOPPED

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 STATE == LO or HI, resp.
SMX_PK_DIFF_CNT	Differential count from timer ahead.
SMX_PK_LSR	LSR to be invoked on timeout.
SMX_PK_MAX_DLY	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_PAR	Parameter value to pass to LSR, if opt = SMX_TMR_PAR.
SMX_PK_PERIOD	Period of cyclic or pulse timer.
SMX_PK_STATE	LO/HI pulse state.
SMX_PK_TIME_LEFT	Total remaining time until timeout for tmr.
SMX_PK_WIDTH	Pulse width of pulse timer.

If timer has already timed out (i.e. tmr == NULL), or par is invalid, returns 0 and reports SMXE_INV_PARM error. If tmr is not a valid timer handle, returns 0 and reports SMXE_INV_TMCB error.

Example

```
TMCB_PTR atmr;
TCB_PTR  task;

smx_TimerStart(&atmr, 5, 10, aLSR, "atmr");
...
task = (TCB_PTR)smx_TimerPeek(atmr, SMX_PK_ONR);
```

In this example, atmr is created. At some later time (and probably by a different task) its owner is determined. The type cast is typical because TimerPeek() returns a variety of parameter types — all as u32. The above will not work if atmr owner is an LSR. Instead:

```
void* hdl;
LSR_PTR lsr;

hdl = (void*)smx_TimerPeek(atmr, SMX_PK_ONR);
if (smx_SysWhatIs(hdl) == SMX_CB_TASK)
    task = (TCB_PTR)hdl;
else
    lsr = (LSR_PTR)hdl;
```

smx_TimerReset

BOOLEAN smx_TimerReset (TMCB_PTR tmr, u32 *tlp)

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	Error. Timer not restarted.

Errors SMXE_INV_TMCB

Descr Dequeues 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. Then requeues timer in tq using its current delay and returns TRUE.

If the timer 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. Then, 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 — i.e. the time until the next timeout.

If the timer has already timed out (i.e. tmr == NULL), returns FALSE and loads 0 into tlp, unless it is NULL. The timer cannot be restarted in this case because its TMCB has already been cleared and returned to the TMCB pool. If tmr is not a valid timer handle, aborts operation with FALSE, loads 0 into tlp unless it is NULL, and reports an SMXE_INV_TMCB error.

Example

```
TMCB_PTR atmr;

smx_TimerStart(&atmr, 10, 0, aLSR, "atmr");
```

```
while (1)
{
    while (wait_for_event()) {}
    /* perform actions */
    smx_TimerReset(atmr, NULL);
}

void aLSR(u32 par)
{
    /* deal with timeout */
}
```

In this example, atmr is started. Then the while loop waits for an event. When the event occurs, it performs the required actions, then resets atmr. If the next event does not occur within 10 ticks, atmr times out and invokes aLSR 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 (TMCB_PTR tmr, LSR_PTR lsr, SMX_TMR_OPT opt, u32 par)

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_PARM
SMXE_INV_TMCB
SMXE_TMR_STOPPED

Descr Loads new values for LSR, LSR option, and LSR parameter into the timer's TMCB. The LSR option controls what is passed to the LSR when it is invoked:

SMX_TMR_PAR	par stored in TMCB.
SMX_TMR_STATE	pulse state (LO/HI).
SMX_TMR_TIME	etime at timeout.
SMX_TMR_COUNT	number of timeouts since start.

These options help to reduce the need for LSR peeks. 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.

If timer has already timed out (i.e. tmr == NULL), lsr == NULL, or opt is invalid, returns FALSE and reports SMXE_INV_PARM error. If tmr is not a valid timer handle, returns FALSE and reports SMXE_INV_TMCB error.

Example

```
TMCB_PTR atmr;

smx_TimerStart(&atmr, 10, 10, aLSR, "atmr");
smx_TimerSetLSR(atmr, aLSR, SMX_TMR_COUNT, 0);

void aLSR(u32 count)
{
    if (count < 100)
        /* perform function */
    else
        smx_TimerStop(atmr, NULL);
}
```

In this example, atmr is started, then it is modified to pass the timeout count to aLSR. After 100 timeouts, aLSR stops further action.

smx_TimerSetPulse

BOOLEAN smx_TimerSetPulse (TMCB_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_PARM
SMXE_INV_TMCB
SMXE_TMR_STOPPED

Descr Loads new values for timer period and pulse width into its TMCB. 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. Or if called in the middle of an inter-pulse period (state == LO), that period is allowed to complete normally. In either case, the new width and new period will be applied starting with the next period. This assures 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 UG Timer sections for more discussion.)

If timer has already timed out (i.e. tmr == NULL) or width >= period, returns FALSE and reports SMXE_INV_PARM error. If tmr is not a valid timer handle, returns FALSE and reports SMXE_INV_TMCB error.

Example

```
TMCB_PTR atmr;

smx_TimerStart(&atmr, 5, 10, aLSR, "atmr");
smx_TimerSetPulse(atmr, 10, 2);
smx_TimerSetLSR(atmr, aLSR, SMX_TMR_STATE, 0);

void aLSR(u32 pulse)
{
    if (pulse == HI)
        Lamp(ON);
    else
        Lamp(OFF);
}
```

In this example, atmr is started, then changed to a pulse timer with a pulse width of 2 ticks and a period of 10 ticks (i.e. 2 ticks HI and 8 ticks LO). The timer is set to pass the pulse state to aLSR when it times out. This is used to turn a lamp on or off. In this example, after the 5 tick start delay, the first 2 tick pulse will occur and it will occur every 10 ticks, thereafter.

smx_TimerStart

BOOLEAN smx_TimerStart (TMCB_PTR *tp, u32 delay, u32 period, LSR_PTR lsr, const char *name)
BOOLEAN smx_TimerStartAbs (TMCB_PTR *tp, u32 time, u32 period, LSR_PTR lsr, const char *name)

Type SSR

Summary Creates and starts a new timer or restarts an existing timer.

Compl smx_TimerStop()

Parameters	tp	Pointer to location for timer handle.	
	delay	Timeout, in ticks, from now.	
	time	Absolute time from startup (i.e. etime == 0).	
	period	Period, if cyclic timer, 0 if not.	
	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_PARM		tp == NULL, delay == 0, or lsr == NULL.
	SMXE_INV_TMCB		*tp is not a valid timer handle.
	SMXE_OUT_OF_TMCBS		

Descr

If *tp == NULL, this call creates a timer and starts it running. A timer control block (TMCB) is allocated from the TMCB pool (smx_tmcbs), and the start parameters: delay, period, lsr, and name are loaded into it. In addition, the TMCB onr field is set to the current task or the current LSR, depending upon which made this call. Other TMCB fields are set to default values, which can be changed by other timer services.

If an existing timer is being restarted (i.e. *tp != NULL), the timer is dequeued from the timer queue, tq. Then the delay, period, lsr, and name fields in the TMCB 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 TMCB, and it is singly-linked into tq. This process is similar to that for an event queue. (See smx_EventQueueCount().) Then its handle is loaded into the location at tp.

The address of the user's timer handle variable is saved in the TMCB 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 TMCB 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.

smx_TimerStartAbs() is identical to smx_TimerStart() except that it accepts an absolute time from system start (i.e. etime), rather than a delay. This is useful to assure 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 User's Guide for an example of using absolute timer starts.

If tp == NULL (NULL pointer reference), delay == 0 or time <= smx_etime (timeout already passed), or lsr == NULL (no action possible), FALSE is returned SMXE_INV_PARM error is reported. If starting a new timer and no TMCB is available, FALSE is returned and an SMXE_OUT_OF_TMCBS error is reported. If restarting a timer and its handle is not valid, FALSE is returned and SMXE_INV_TMCB error is reported. In this case the intended timer will continue running as if nothing happened.

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. If time left is needed, it is recommended that the timer be stopped then restarted.
3. Failure to restart a running timer, due to an error, does not stop it.
4. tp points to a slot used by a task used to store the handle of a timer that it started. This slot should must not be used by any other task, else errors will occur due to one task restarting another task's timer.

Examples

```
TMCB_PTR  atmr;

smx_TimerStart(&atmr, 10, 0, aLSR, "atmr");

void aLSR(u32 par)
{
    /* perform timeout function */
}
```

The above example shows creating a one-shot timer that invokes aLSR to perform a timeout function after 10 ticks. This occurs only once, and atmr deletes itself.

```
smx_TimerStart(&atmr, 10, 10, aLSR, "atmr");
```

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 (TMCB_PTR tmr, u32 *tlp)

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_TMCB

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 handle is automatically cleared so that the timer cannot be accessed again. (This was the location provided by smx_TimerStart().) After stopping a timer, its TMCB is cleared and returned to the timer pool.

If tmr == NULL, 0 is loaded into *tlp, and TRUE is returned. The condition occurs if attempting to stop a timer that has already stopped, been stopped, or never started. If tmr is not a valid timer handle, FALSE is returned and SMXE_INV_TMCB error is reported.

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

```
TMCB_PTR  atmr;
u32  time_left;

smx_TimerStop(atmr, &time_left);
```

This shows stopping the timer started in the example for smx_TimerStart(). The total time left for the timer before it would have timed out is stored in time_left. This might be useful, for example, to determine if shorter or longer timeouts should be used.

smx Glossary

This section briefly defines smx terminology and refers to other documents for details. Terms are in alphabetical order. When searching for a term, note that the smx_ or SMX_ prefix is generally omitted (else nearly all entries would be under “s”). So, for example, look for “ct”, not “smx_ct”. The main exception to this is that errors are all listed under “SMXE_” in order to keep them together.

The entries in this glossary often have a more implementation details than text in other places and thus may help to give a better understanding of the entries.

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 allocated.

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.

access conflict Occurs when two routines try to simultaneously access a non-sharable system resource. Access conflicts due to preemption are very similar (and equally troublesome) to those caused by hardware interrupts. It helps to use data hiding as much as possible and to use messages and pipes to transmit data from task to task, rather than to use global variables. See also: critical section and lock.

accurate With respect to timing, means accurate to a tick.

active task A task which is running or suspended, but not stopped.

ADAR **application dynamically allocated region**. One of two DARs created by smx. This DAR contains the stack pool, usually the heap, and user-defined block and message pools. It can be used for other dynamic user objects. See UG Memory Management.

AND **== SMX_EF_AND**. Bit 16 set in an event flags mask.

ANDOR **== SMX_EF_ANDOR**. Bit 17 set in an event flags mask. Overrides AND (bit 16).

atomic Indivisible. As applied to software means that a group of statements, cannot be interrupted by other software, which shares variables or resources. Generally, SSRs are atomic. To protect a task’s code from other tasks, lock the task; to protect from LSRs disable LSRs; and to protect from ISRs, disable interrupts.

amerge mode is controlled by the **smx_heap.mode.amerge** flag. It starts ON (except if MIN_RAM) and can be turned OFF or ON via smx_HeapSet(). In the ON state, automatic merge control is enabled. Example code for this is in smx_HeapManager() in main.c of the Protosystem. See discussion in UG Heap Management. In the OFF state, chunk merging can be manually controlled.

automatic merge control See above.

autostop Returning from a task's main function or running through the last brace ("}") results in an autostop. If this occurs, the task is stopped forever and must 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.

base block is a data block from an smxBase block pool. It is obtained with sb_BlockGet(). See SB.

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 UG Memory Management, base block pools and SB Base Block Pools.

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. See bare function description above.

BCB **block control block** There is one BCB per smx block. It contains a block pointer, pool handle, and owner (task or LSR) handle. A BCB is free if its owner == SMX_ONR_NONE is free.

BCB pool All BCBs are in a pool, which is controlled by the smx_bcbs pool control block. The singly-linked list of free BCBs is pointed to by smx_bcbs.pn. The link pointer is in the first word of each free BCB. The last free BCB has a NULL link.

BCB_PTR BCB pointer type. Variables of this type contain smx block handles.

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, this will be the first large-enough chunk found in the bin.

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. Also occurs when cmerge is OFF and chunks are split and remnants are moved to smaller bins.

bin-type heap A heap that uses bins to store free chunks. 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. See also UG Semaphores.

block A block is a group of adjacent memory locations. A block may be any size. There are many types of blocks: smx blocks, message blocks, base blocks, DAR blocks, heap blocks, static blocks, etc. See descriptions in UG Memory Management and SB Base DAR Functions and Block Pools.

block migration refers to the process of making a block into a message to pass it to the background or unmaking a message into a block to pass it to the foreground. See UG Exchange Messaging

block pool A pool of equal-size blocks controlled by a pool control block. See base block pool and smx block pool.

bmap **bin map** has one bit per bin. If the bit is set, the bin contains at least one chunk.

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 mode If a task has a stack, it is in the bound mode and, if not, it is in the unbound mode. If the task's stk_perm flag is set, it has a permanently bound stack. Otherwise, the stack is bound to the task by the scheduler when the task is dispatched.

Normal tasks are always in the bound mode. One-shot tasks are created in the unbound mode and vary between bound and unbound depending upon whether they are running/suspended or stopped.

broadcasting consists of sending one message to a broadcast exchange. Any tasks receiving from the exchange will receive the message handle and data pointer. However, they can only read the message. See **slave task** and **master task**.

bs_fwd mode is controlled by the **smx_heap.mode.bs_fwd** flag. It starts ON and controls the direction of heap bin scans. It is an internal mode and is not user controlled.

bsmap **bin sort map** has one bit per bin. If the bit is set, the bin needs to be sorted.

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 associated with each.

The format of the first 3 fields is common to all control blocks, except BCBs and PCBs. All have a forward link, backward link, and control block type. Control blocks are created and initialized by the Create functions, such as smx_TaskCreate(). One control block is required per object; it is assigned when the object is created. Control block handles are used to access control blocks. See handle for more information.

Control blocks are cleared when released so that new control blocks are always empty. This is not only safer but it is clearer when looking at them while debugging, since it is obvious whether it is in use or not.

For further information on a specific control block type, look in this glossary under the cbtype or subtype name given above. All smx control block types are defined in xtypes.h.

cbtype **control block type**. This field is present in nearly all control blocks; it is always in the same position, if present. Values are defined in xdef.h, for example SMX_CB_TASK.

CCB **chunk 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 allocated block problems. A CDCB contains 24 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 must be at least = block_size + CHK_OVH.

chunk A block of memory used by the heap. A chunk consists of a header 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.

cmerge mode is controlled by the **smx_heap.merge.cmerge** flag. Normally it starts OFF. In this state, chunk merging by smx_HeapFree() is inhibited, which helps to populate heap bins. When ON, merging of chunks is enabled. This helps to avoid allocation failures by reducing fragmentation. Can be turned ON or OFF via smx_HeapSet(). See amerge mode for automatic control.

client task is provided a service by a server task. Typically, a client task sends a message or proxy message to a message exchange, then waits for a response. See UG Exchange Messaging.

clsr **current LSR**. Address of current LSR or 0 if no LSR is running. While an LSR is running, the current task is effectively suspended. Stored in smx_clsr.

cmerge **smx_heap.mode.cmerge.** Starts OFF. Can be turned ON or OFF by smx_HeapSet(). When ON, chunks are merged with adjacent free chunks when freed. When OFF chunk merging does not occur.

complementary call The smx call that performs the inverse operation of a particular call e.g.: smx_MsgSend() vs. smx_MsgReceive(), and smx_SemSignal() vs. smx_SemTest().

complementary pipe function An smx pipe function that may be used at the other end of the same pipe.

configuration table See smx_cf below.

context switch Same as task switch.

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, which is a statically allocated smxBase PCB. Control block pools are allocated from SDAR, when first needed (usually by the corresponding create function). See specific pool types and UG Under the Hood, dynamic control blocks, for further information.

counting semaphore is the same as a **resource semaphore**.

coupled ISR See smx ISR.

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 preemptions. See atomic.

ct **current task**. is the currently running task. It handle is stored in smx_ct. smx_ct == NULL should occur only during startup, since the smx_Idle task should always be ready to run. Do not use smx_ct in ISRs or LSRs since the current task may have already been removed from rq and enqueued elsewhere.

current delay **for a timer —** If the timer 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. Then, 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 — i.e. the time until the next timeout.

DAR **dynamically allocated region** — A region of memory for dynamically allocated structures. smx ships with SDAR for smx objects and ADAR for application objects. Additional DARs can be created for special purposes. See SB Dynamically Allocated Regions.

data abort An exception due to accessing unimplemented memory.

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 system failure may occur, in a hard real-time system.

deadlock or deadly embrace occurs when two tasks are waiting upon resources owned by each 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 and ceiling priority. Timeouts serve to break deadlocks.

debug chunk is a heap chunk, which is currently inuse and which contains debug metadata. The debug metadata consists of a Chunk Debug Control Block, CDCB and fences surrounding the data block. The number of fences is user-specified.

debug mode is a flag in **smx_heap.mode.debug.** It starts OFF and can be turned ON or OFF via smx_HeapSet(). When ON, allocations produce debug chunks; when OFF, allocations produce inuse chunks.

debug version The version of the smx library, 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 from a queue. This is done by changing forward and backward links (fl and bl) in appropriate control blocks. 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.

dispatch Dispatching a task is the process of starting it to run. This is done by the scheduler.

For a one-shot task, it is necessary to get a stack from the freestack pool. The scheduler then starts the task by calling its main function, with tcb.rv as its parameter. If a stack is not available, SMXE_OUT_OF_STKS is reported and the scheduler goes on to the next ready task. This error is reported only once. The task remains in rq until a stack is available.

The first time a normal task is dispatched, it is started like a one-shot task, except it already has a stack. When a task has been suspended, the scheduler resumes it by returning past the point of suspension. This requires restoring registers and passing back the return value if an SSR was called.

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 the blank message body. See UG Exchange Messaging, broadcasting and proxy messages.

donor chunk is located between the lower heap and the upper heap. Initially it is located immediately after start chunk. It supplies small chunks for the lower heap, which are of SBA size. If the SBA bin for the desired size is empty, the chunk is taken from dc. This helps to maintain locality of SBA chunks. dc must be large enough to hold a chunk control block (CCB).

dormant A task is dormant if it is stopped with no timeout. Such a task will not run again unless it is started by another task.

double free occurs when free() attempts to free a chunk, which is already free. If the chunk has not already been reallocated, this is detected and SMXE_HEAP_ERROR is reported.

dynamic A dynamic object can be created and deleted, at run time. All smx objects can be dynamically created, and all, except DAR block pools, can be dynamically deleted. See also static.

dynamic merge control Control of heap chunk merging that is implemented on a task or function-specific basis.

dynamically allocated region See DAR.

EB See **error buffer**.

ec See **end chunk.**

EG **event group** consists of event flags and masks. Each event flag is one bit in a 14-bit field in an EGCB, which can be set or reset by service calls. Each mask defines an AND, OR, or AND/OR combination of the event flags. Each waiting task has its own mask. When a match occurs, due to setting a flag, all tasks with matching masks are resumed. See UG Event Groups.

EGCB **event group control block** controls an event group. It contains forward and backward links for the task wait queue, flags, and the event group name..

EGCB pool See QCB pool.

EGCB_PTR EGCB pointer type. Variables of this type contain event group handles.

EM **smx_EM()**. See Error Manager.

EMHook smx_EMHook() is a callback function for the user to add error processing code inside of smx_EM(). Can also be used to set an error breakpoint. See UG Error Manager, error manager hook.

end chunk is the last chunk in the heap. It is an 8-byte, inuse chunk with no data block. smx_heap.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 is a layered priority queue — see ready queue. Enqueueing is a logical process. Control blocks are not moved — all stay in the same physical location.

entry routine A hooked entry routine, which is transparently called by the scheduler prior to resuming a task. See hook routine.

EQCB	**event queue control block** controls an event queue. It contains forward and backward links for the task wait queue, tq flag, and the event queue name.
EQCB pool	See QCB pool.
EQCB_PTR	EQCB pointer type. Variables of this type contain event queue handles.
EREC	**error record format** for the error buffer (EB) contains fields for etime, error number, and a handle identifying the source of the error.
error buffer	**smx_EB** is an array of error records stored cyclically — the oldest is overwritten by the newest. EB is allocated from SDAR and cleared during initialization.
error manager	**smx_EM()**. The smx error manager is called whenever an error is detected by smx. It is usually called via the smx_ERROR() or smx_ERROR_EXIT() macros in SSRs. It updates the err globals below, as well as tcb.err for the current task. If enabled, it saves information in EB and EVB, and displays an error message on the console. For most errors, control then goes back to the point of call with a failure indication. Severe errors may lead to restarting the current task or exiting the application. See UG Error Management.
error type	smx error types are defined in SMX_ERRNO in xdef.h. An enum is used for compilers which permit byte enums; for other compilers, defines are used. There are nearly 70 types of errors.
errctr	**smx error counter**, stored in smx_errctr. Counts all errors since system startup. sb_errctr is the corresponding variable in the smxBase error manager.
errctrs	**smx error counters** array, stored in 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 error number**, stored in smx_errno. Stores the error number of the last smx error detected. To determine the last error caused by a particular task, consult task->err, instead. Error numbers are defined in the SMX_ERRNO enum in xdef.h. sb_errno is the corresponding variable in the smxBase error manager. Keep in mind that some smx services use smxBase services, so the smxBase service may be the actual cause of failure. Consult sb_errno when debugging, if it is unclear what is the cause of an smx error. Note: SMX_ERR will register that an error has occurred.
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 >= 2exp31. When a rollover occurs, the top bit of etime and all active timeouts is cleared. This is performed in the idle task, with LSRs turned off so that smx_KeepTimeLSR() and smx_TimeoutLSR() (which perform all timing functions) cannot run.
EVB	See **event buffer**.
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 In an event group, setting an event flag signals that an event has occurred. This may cause a match and result in one or more tasks being resumed and the event flag being reset. See UG Event Groups.

event queue An event queue permits a task to be resumed or restarted after a specified number of events have occurred. Tasks are enqueued in order, by differential counts. Signals decrement the first task's counter until it is zero. Then it is resumed or restarted. If the next task's counter is zero it also is resumed or restarted. If not, it is decremented by subsequent signals. Examples of events are: alarms, rotations, triggers, pulses, etc. A task using an event queue will miss events while it is not enqueued in the event queue (i.e. when it is running or ready to run). If this is not acceptable, a semaphore should be used instead. See UG Event Queues.

exchange An **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.

These operate similarly. See descriptions of each type, below and UG Exchange Messaging.

exchange messaging a messaging technique wherein messages are exchanged between senders and receivers via exchanges. See UG Exchange Messaging.

exit routine A hook routine, which is transparently called by the scheduler when suspending a task. See hook routine.

external fragmentation In a memory management system, external fragmentation refers to wasted space due to blocks being smaller than useful or more blocks in a pool than are ever needed.

FALSE 0 or !TRUE. See BOOLEAN.

fence is a known pattern, such as 0xAAAAAAA3, in a debug chunk before and after the data block. The pattern is determined by SMX_HEAP_FENCE_FILL in xcfg.h. This can be any pattern as long as bits 1 and 0 are 1's. The number of fences after the data block is SMX_HEAP_FENCES (xcfg.h) and before is SMX_HEAP_FENCES + 1.

fill mode is controlled by the **smx_heap.mode.fill** flag. It starts OFF and 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 they are not.

flyback There are two 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 run another task if higher priority. This is done to minimize LSR and task latencies.

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 stack. This is especially likely if they are called via a software interrupt. Stack checking must be inhibited while using a foreign stack, using smx_TaskSetStackCheck().

fragmentation as applied to a heap means that chunks become smaller and smaller and thus less useful. Severe fragmentation may result in failure to be able to allocate larger chunks.

frame Relative to smx, means the profile frame used to capture profile information. The length, in ticks, is set by RTC_FRAME in acfg.h. See UG Precise Profiling.

free() Generic heap free operation that frees inuse chunks to the heap.

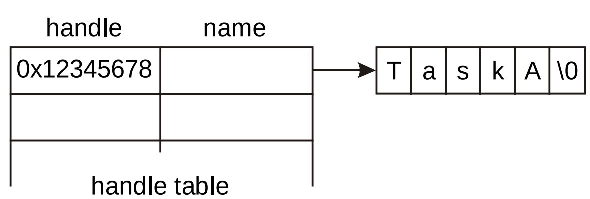
free chunk A heap chunk that is not in use and thus free to be allocated. A free chunk consists of a 24-byte Chunk Control Block, CCB and free space.

free chunk list Doubly-linked list of free chunks in a heap bin. Free forward links (ffl's) and a 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 For a gate semaphore, all waiting tasks are resumed by one signal. See UG Semaphores.

handle In an smx system, the handle of an object is the address of its control block. A handle is stored in a variable of type _CB_PTR (e.g. TCB_PTR), which should be named after the object (e.g. atask). Handles are returned by smx calls, which create objects, and they are passed to other smx calls, which operate on objects. Handles are generally treated as object identifiers, not as pointers, since they are used only as arguments for smx services.

handle table Stores handles and pointers to names. Used by smxAware. Handles are added by smx_HT_ADD(); they are removed by smx_HT_DELETE(). The following diagram shows the handle table structure.



hard real-time means that a system failure may occur if a deadline is not met.

HCB **heap control block** is a static control block, which controls the heap**.** It contains a pointer to the start chunk of the heap, pi, a pointer to the end chunk of the heap, px, the heap mode, and the heap name.

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 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 bin A heap bin is the head of a free list of doubly-linked chunks of a certain size or of a certain range of sizes.

heap block is a data block allocated from the heap. It is contained within a chunk.

heap 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 many chunks may be bridged over.

HEAP_CZMAX Configuration constant in acfg.h used in auto-merge control. It defines the maximum dynamic chunk size expected for the application, excluding one-time allocations on startup.

If neither the largest chunk in the top bin nor the top chunk are at least this large, cmerge is turned ON.

heap fence A one-word pattern defined in xcfg.h as SMX_HEAP_FENCE_FILL. Fences surround a data block in a debug chunk. There are SMX_HEAP_FENCES, defined in xcfg.h, above the data block and SMX_HEAP_FENCES +1 below the data block. The fence pattern can be changed, but bits 1 and 0 must be 1. (These are the *alternate* DEBUG and INUSE flags.)

heap range test is a test of a chunk pointer to verify that it is within the range of the heap, i.e.: smx_heap.pi <= cp <= smx_heap.px. eheap tests all pointers, before use, in order to find broken pointers and to avoid data abort exceptions. If the heap has been extended over a gap, this test will be less effective. Note: If SMX_HEAP_SAFE is not OFF, some heap range tests are disabled for speed.

heap stack A stack taken from the heap, if stack size is not 0 in smx_TaskCreate(). A heap stack is permanently bound to a task and remains bound until the task is deleted. The stk_perm flag in a task's TCB, if set, indicates if the task has a permanent stack.

HEAP_TZMIN Configuration constant in acfg.h used in auto-merge control. If the sum of chunk sizes in the top bin (smx_tbsz) plus the top chunk size is less than this value, cmerge is turned ON. See UG Heap Theory, deferred merging for more details.

high-water mark is the maximum number of bytes of stack or heap used since startup. Each task stack high-water mark is saved in task->shwm. The system stack high-water mark is saved in smx_Nulltask->shwm. The heap high-water mark is saved in smx_heap_hwm. These values are displayed in smxAware and can be used to tune stack and heap sizes.

hook routines are routines called by the scheduler for a hooked task. The hook exit routine performs task-specific functions when the task is preempted or suspended, and the hook entry routine reverses these functions when the task is resumed. Typically, this feature is used to extend the task context, to save additional data on suspend and to restore it on resume. Hook routines can be used for other purposes, such as time measurement. Exit and entry routines must preserve non-volatile registers that they use

host system Refers to the development system on which application software is edited, compiled, and linked and which runs the debugger

hs_fwd mode is controlled by the **smx_heap.mode.hs_fwd** flag. It starts ON and controls the direction of heap scans. It is an internal mode, not user controlled.

ht See **handle table**.

idleup Indicates that idle has been boosted temporarily 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. Stored in smx_idleup.

init mode is controlled by the **smx_heap.mode.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, internal fragmentation refers to wasted space due to a chunk 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 in smx. 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 8-bytes of metadata plus the data block being used by the application.

ISR **interrupt service routine**. A function which handles a hardware or software interrupt. An ISR is usually invoked via a vector stored in an interrupt vector table (IVT), however various mechanisms are used by various processors. An smx ISR is one that invokes 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 they do not enable interrupts. ISRs cannot call smx services. other than smx_LSR_INVOKE() and bare pipe functions. See UG Service Routines.

large bin A heap bin that stores a range of chunk sizes, which are 8-byte aligned and multiples of 8 bytes.

large chunk A large chunk is one that fits into an upper bin.

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.

least-big-enough Means to find the smallest free chunk in the heap that is big enough for the requested block size. This is also called the best-fit chunk.

limited SSR An smx service that can only be called from tasks not LSRs. These are primarily SSRs that would stop or suspend the current task as a side effect of their running.

linear address In a system with the memory management unit (MMU) disabled, the linear address is equal to the physical address. When the MMU is enabled (i.e. paging is enabled), the linear address is translated into a physical address by the MMU.

linear heap has only a physical structure and must be searched sequentially to find large-enough chunks to allocate.

link service routine See **LSR**.

localization As applied to heaps, means to allocate chunks, which are close in time, to be physically close in order to increase cache hits, assuming the chunks are being used by the same task.

locked A task is locked if smx_lockctr > 0. When locked, a task cannot be preempted. See UG Tasks.

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.

lower bin array That portion of the smx_bin[] array that contains the small bin array, SBA, also known as *lower bins*.

lq **LSR queue** is a queue which contains the address and parameter of each invoked LSR, in the order invoked. lq is cyclic. If it overflows (i.e. a newly invoked LSR overwrites one that has not yet executed), SMXE_LQ_OVFL is reported.

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 and 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. For more information, see UG Service Routines.

LSR_PTR is the standard LSR format: void lsr(u32 par). LSRs never return any value. The LSR parameter can be defined as a different type:

```
void  msg_LSR(MCB_PTR msg);
```

but when invoked both must be typecast:

```
smx_LSR_INVOKE((LSR_PTR)msg_LSR, (u32)msg);
```

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 scheduler calls when it starts the task. Its address is stored in tcb.fun.

malloc() Generic name for all heap allocation services.

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 slave task and UG Exchange Messaging, broadcasting messages.

MCB **message control block.** Each active smx message has an MCB, which contains important message parameters. 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 **MCB pointer** type. Pointers of this type store message handles.

memory leak is loss of usable memory. This normally occurs in a heap due to failure to free blocks when no longer needed. Reallocating the blocks, when needed again, results in steady loss of free heap space.

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 to transfer data and control information. smx_MsgGet() gets a data block from an block pool and links it to an MCB from the MCB pool. smx_MsgRel() reverses this process. See UG Exchange Messaging.

message body same as message data block.

MIN_FRAG Configuration constant in xheap.c or eheap.c 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 accumualtion of unusable chunks in lower bins.

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. See UG Event Groups.

MUCB **mutex control block** controls a 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 it is released.

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 **MUCB pointer** type. Pointers of this type store mutex handles.

macro smx macro names are all caps, except 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. smx multiline macros are enclosed with {} in their definitions, so they can be called like functions.

message queue of other kernels is the same as an smx pipe used for intertask communication. See UG Pipes.

migration See **block migration.**

multicasting consists of sending proxy messages to multiple exchanges. This provides more control than broadcasting. See UG Exchange Messaging, proxy messages and multicasting.

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 can have two states: free and owned. Only one task at a time can own a mutex. See MUCB and UG Mutexes.

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-volatile registers 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.

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(). See also UG, Service Routines, two ISR types.

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 not task is waiting, the message is enqueued at the exchange and given to the first task that receives a message from the exchange. See UG Exchange Messaging.

NULL Means a null pointer — i.e. 0. We generally try 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 (or simply, objects) consist of tasks, pools, blocks, messages, exchanges, queues, etc. created by an application. smx++ provides system objects from which application objects may be derived. See SPP.

smx objects consist of control blocks, variables, and constants used by smx to control the system. Generally, smx handles these objects, and the user need not be concerned with them.

one-shot task is a task which stops when done and releases its stack. Hence, one-shot tasks can greatly reduce RAM requirements in systems having many tasks. A one-shot task is created with 0 for the stack size parameter. See UG One-Shot Tasks.

pass exchange This type of 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. See UG Exchange Messaging.

OR **== SMX_EF_OR**. OR condition in an event flag mask. Means bits 16 and 17 are both zero. Not actually required, but used for clarity of intention.

PCB **pool control block** controls an smx block pool. It has pointers to the first and last blocks of the pool, a free block list pointer, block size, and the number of blocks in the pool, and the pool name.

PCB pool All PCBs are in a pool, which is controlled by the smx_pcbs pool control block. 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 **PCB pointer** type. Pointers of this type store pool handles.

permanent stack is a stack that is bound to a task when the task is created. Permanent stacks come from the heap, and unlike a temporary stack, a permanent stack remains bound to the task even if it stops. It is only released when the task is deleted.

physical address is the address actually placed on a memory bus. Unless a memory management unit (MMU) is used for paging, physical addresses and linear addresses are the same.

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 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. The flags must be stripped from blf before using it as a pointer to the previous chunk.

PICB **pipe control block** controls a 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 **PICB pointer** type. Pointers of this type store pipe handles.

pipe An smx object which permits transfer of bytes or packets between tasks and between tasks and ISRs or LSRs. Packet size is determined when the pipe is created and may be 1 to 255 bytes. Tasks and LSRs use put wait SSRs to put characters into pipes and get wait SSRs to get characters from them. ISRs use put8 and put packet functions and get8 and get packet functions, respectively.

A pipe is empty when the read and write pointers are equal. A pipe is full when the write pointer is exactly one cell behind the read pointer. Put and get SSRs result in tasks waiting on pipes when an operation cannot be completed. Put and get functions, which are intended for ISR usage, do not wait and also cannot resume or restart a waiting task. See UG Pipes.

pool A pool consists of contiguous blocks of equal size. smx uses base block pools (see SB Base Block Pools) for all smx control blocks. See block pool for smx data blocks. See UG Memory Management for more discussion of pools.

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. Generally, however, it is much more precise than a tick.

precise profiling See **profiling.**

preemptible An smx task is preemptible if it is not locked. Preemption can only be performed by a higher priority task.

porting layer A set of functions, macros, and defines that isolate the characteristics of a library that change from one target and operating system to another. The goal of a porting layer is to be able to port a library to another environment by changing only the things in the porting layer, leaving the bulk of the code unchanged.

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 (i.e. an interrupt), 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. Hence care must be taken to protect critical sections in tasks.

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. 0 is the lowest priority and 126 is the highest priority. All tasks and messages have priorities. An enumerated data type is a good way to define priorities. For example:

```
typedef enum {PRI_MIN, PRI_LO, ...} PRIORITIES;
```

This provides good readability and allows a new level to be easily added. This enum with 8 levels has already been defined in xcfg.h. These levels are used by SMX middleware. See UG Tasks, task priority.

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 should equal the highest priority of any task that may own the object. This avoids unbounded priority inversion and also eliminates deadlocks for objects having the same ceiling. See UG Mutexes.

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 provide for propagation from one mutex to another mutex when the owner of the first is waiting for the second. Propagation can cover any number of mutexes. smx mutexes also implement staggered priority demotion when a mutex is released. See UG Mutexes.

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 one or more mid-priority tasks. The resulting delay of the higher priority task is unpredictable and may cause it to miss a deadline. smx provides priority ceiling and priority inheritance to deal with this problem.

process A process is usually an independent program loaded by an operating system from disk and consisting of many threads (AKA tasks). smx does not support processes.

processor architecture is frequently used to refer to header and other files. Typical processor architectures are ARM, ARM-M, ColdFire, etc. Within an architecture there are different processor families, which may be made by different vendors. In addition, there are different tool sets. For your delivery or evaluation kit, you need to deal with only one combination of these.

However we must deal with a hundred processor families and half a dozen tools — a vastly more complex endeavor. This results in fairly deep directory trees, for example: BSP\ARM\AT91\SAM9\AT91SAM9G20EK\Atmel. The last level contains vendor files for the specific processor. The next level up contains files specific to AT91SAM9G20EK, such as boot_gcc.s or boot_iar.s. The next level up contains files specific to SAM9 such as bsp.c. This process is repeated up to the top level. The reason for spreading files throughout a directory tree is to minimize file duplication — files are put at the level where they cover all combinations below.

Due to differences between processors, it has not been possible to adapt a uniform naming system. Hence, we often refer to "the processor architecture header file", for example. Hopefully, in your specific instance, you can figure out what that means. See TG for more information.

profiling smx provides precise and coarse profiling. Precise profiling records run time counts (RTCs) in the TCBs of all tasks and also run time counts for all ISRs, combined, and all LSRs, combined. Counts are accumulated for a frame, which can be any number of ticks from 1 to about 68,000, then loaded into the RTC buffer, which cyclically stores 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 UG Precise Profiling.

Protosystem An smx mini-application that serves as a foundation for application development and allows running the demos supplied with smx products. See QS Protosystem.

proxy message consists of an MCB which points to a real message data block. A proxy message can be made from a real message with smx_MsgMake(NULL, dp), where dp points to the real message data block. Proxy messages are used for multicasting and distributed message assembly.

PSS Pipe Status Structure. Used to store pipe width, length, flags, number of packets, and number of waiting tasks. Loaded by an smx_PipeStatus() call.

ptime means **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. See SB BSP/ Time Functions.

QCB **queue control block.** smx has four types of queue control blocks:

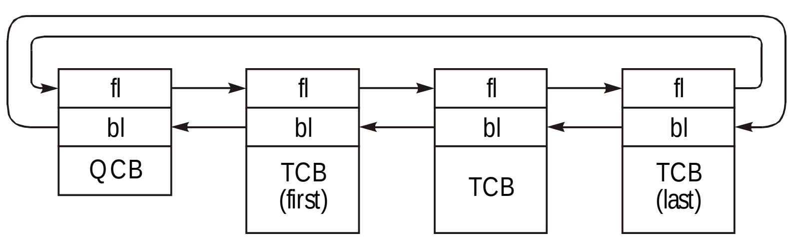
EGCB	Event group.
EQCB	Event queue.
SCB	Semaphore.
XCB	Exchange.

All queue control blocks are in the QCB pool and have similar formats. See each for more information.

QCB pool All QCBs are in a pool, which is controlled by the smx_qcbs pool control block. The singly-linked list of free QCBs is pointed to by smx_qcbs.pn. The link pointer is in the first word of each free QCB. The last free QCB has a NULL link.

QCB_PTR **QCB pointer** type. Pointers of this type store queue handles.

queue Most smx queues are doubly linked lists of tasks or messages as shown in the following example:



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 its bl is changed to point to the QCB. Then, the first TCB is free to be linked into another list, such as the ready queue. Message queues are the same, except that MCBs are linked in instead of TCBs. Most smx queues are headed by QCBs of various types.

The smx objects which may contain a task or a message queue are as follows:

object	name	queue types
SMX_CB_EG	event group	task FIFO
SMX_CB_EVQ	event queue	task count
SMX_CB_NXCHG	normal exchange	task or msg priority
SMX_CB_PIPE	pipe	task priority
SMX_CB_PXCHG	pass exchange	task or msg priority
SMX_CB_RQ	ready queue	task multi-level priority
SMX_CB_SEM	semaphore (!gate)	task priority
SMX_CB_SEM	semaphore (gate)	task FIFO
SMX_CB_TQ	timer queue	timer count

queue messaging of other kernels is the same as pipe messaging by smx. See UG Pipes.

ready queue holds tasks that are ready to run. It consists of one level per priority, starting at 0 and going up to the SMX_MAX_PRIORITY specified in acfg.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 level (0) is used by smx_Idle and can also be used for other low priority tasks. It is the level for time-sliced tasks, if SMX_CFG_TIMESLICE != 0.

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. See UG Exchange Messaging.

register save area (RSA) is the area below a stack block 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. xarm.h), since it is processor dependent.

release version is the version of the smx library, middleware, or application intended to be embedded in the shipped product. It is compiled at high optimization, with debug symbolics disabled, and SMX_BT_DEBUG undefined. For an application, the release version is typically compiled at high optimization like the ROM version, but intended to run from RAM, under the debugger. See also: debug version and ROM version.

remnant The remainder of a chunk after splitting a chunk. It must be at least MIN_FRAG (xheap.c) bytes or the initial chunk will not be split. It will always be above the allocated chunk. If the chunk above it is free and cmerge is ON, it will be merged with it.

reply smx_MsgSendPR() has a reply parameter. It is a QCB handle. Its index is stored in the MCB of the message being sent. This allows the receiving task to respond to the QCB type (e.g. send a message to an exchange or signal a semaphore). This is useful for client / server designs.

resource In a multitasking system, the term resource is generally used to mean something that tasks must share, such as a peripheral, data in memory, or a subroutine. Resources normally must be protected with smx objects such as semaphores or mutexes. See UG Resource Management..

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. See also semaphores.

response time The time from the occurrence of an interrupt until an ISR, LSR, or task begins running to process the interrupt event. ISR response time is governed by interrupt latency, including that caused by other 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. Of course disabling interrupts, turning LSRs off, and locking the current task add to response times.

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 or resume from another task or LSR. Whenever a task is stopped, it cannot resume, it must restart. See UG Tasks, starting and stopping tasks and UG One-Shot Tasks.

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.

RM **smx Reference Manual** — abbreviation used in citations. Usually followed by the name of an smx service.

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 for ROM. 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 over and over. This is normally a cooperative scheduling algorithm, in which running tasks voluntarily yield to allow the next task run. This 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 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. Stored in smx_rqtop.

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 that 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 hooked exit and entry routines to save extended contexts. See hook routines.

SBA See **small bin array.**

SB_DATA_ALIGN Minimum alignment for 32-bit data writes. Typically 4 bytes. See processor architecture header file (e.g. barm.h).

SB_TICKS_PER_SEC Ticks per second. Defined in bsp.h in each BSP.

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 barm.h. We use 0xCDCDCDCD, 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 name.

SCB pool See QCB pool.

SCB_PTR SCB pointer type. Variables of this type contain semaphore handles.

sched An internal smx flag, stored in smx_sched, 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 preemption (anything higher priority to run?)

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. 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 normally the next task dispatched. It also gets stacks for one-shot tasks. If none is available, it skips over the one-shot task, until a stack is available, and runs the next task in rq.

The scheduler runs with interrupts enabled, except briefly, in a few places. This necessitates flybacks to insure 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. They are scheduled from the LSR queue in FIFO order. 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.

SDAR **system dynamically allocated region**. One of two DARs created by smx. This DAR contains smx objects such as control blocks, rq, lq, etc. It is recommended that SDAR be located in on-chip SRAM for best performance and that it be isolated from application memory for reliability. See UG Memory Management.

semaphore Semaphores are used for resource management and event signaling. smx support 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, which resumes or restarts the top waiting task or tasks. See UG Semaphores.

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 UG, Resource Management, server LSRs 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 indicated 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 also: UG Resource Management

service routine smx provides three types of service routines:

ISR	Interrupt service routine
LSR	Link service routine
SSR	System service routine

Unlike subroutines, which are linked by the linker at fixed places in the code, service routines are managed by smx and tend to occur asynchronously. See also ISR, LSR, SSR, and UG Service Routines.

shared stack A stack from the stack pool, which is shared by one-shot tasks.

signal is sent to a semaphore or to an event queue to signal that an event has occurred.

slave task A task which receives messages from a broadcast exchange. A slave task does not get control of a broadcast message. It gets only its handle and message body pointer and normally is permitted only to read the message. See master task and UG Exchange Messaging, broadcasting messages.

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. 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 smx_bin[] array, starting at size 24 and consisting of consecutive bin sizes that are multiples of 8 (e.g. 24, 32, 40, ...) up to the top SBA bin. SBA bins can be accessed very quickly by converting the desired block size to an index into the SBA (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. They can be sent between tasks and LSRs via pipes to transfer data and control information. However, they are normally uses for local task storage. smx_BlockGet() gets a data block from an block pool and links it to an BCB from the BCB pool. smx_BlockRel() reverses this process. See UG Memory Management.

smx_bfp Bin fix pointer used by smx_HeapBinScan() to point to the starting chunk for the next backward run.

smx block pool A data 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. See UG Memory Management, smx block pools.

smx_bsp Bin scan pointer used by smx_HeapBinScan() to point to the starting chunk for the next forward run.

smx call same as an smx service. Can be an SSR, function, or macro. See the smx Calls section of this manual.

smx_cf **smx configuration table** is defined in xtypes.h and is initialized in main.c from user-defined configuration constants in acf.h. smx_cf contains the configuration constants most frequently changed. By putting them in this table, it is not necessary to recompile smx in order, for example, to change NUM_TASKS. This is helpful during debug and also permits shipping evaluation kits with smx in object form. It is recommended that this table be located in ROM for shippable software.

smx_clsr smx global variable that stores the address of the current LSR or 0 if not in an LSR. See clsr.

smx_ct smx global variable that stores the handle of the current task. See ct.

SMX_ERR Status from the last smx service for this task. If == SMXE_OK, the return value is valid; if == SMXE_TMO, a timeout has occurred and the return value is not valid; otherwise an error has occurred, SMX_ERR is the error number and the return value is not valid.

smx_heap_hwm heap high-water mark = the largest value of smx_heap_used since the heap was last initialized.

smx_heap_used The total heap space currently allocated, including chunk overhead.

smx_hfp Heap fix pointer used by smx_HeapScan() to point to the starting chunk for the next backward run

smx_hsp Heap scan pointer used by smx_HeapScan() to point to the starting chunk for the next forward run.

SMXE_ABORT An irrecoverable error has occurred. smx_exit() is called to shut down the system.

SMXE_ABORT_TASK An irrecoverable error has occurred for a task. smx_EMExitHook() is called to allow the user to stop the task, shut down the system, or whatever is appropriate.

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 (i.e. out of range) link (fl or bl) or an unexpected cbtype is found while tracing a queue. Using a broken queue is hazardous because enqueueing or dequeueing objects on it can cause writes to wrong memory locations. Hence, 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(). Our implementation exits the application, but you may wish to change this.

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 does not match the number of locks due to a programming error or unexpected execution path.

SMXE_HEAP_BRKN smx_HeapScan()cannot fix the heap or smx_HeapBinScan() cannot fix a bin 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 to deal with the problem.

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 overflow.

SMXE_HEAP_FIXED smx_HeapScan() has fixed a heap problem or smx_HeapBinScan() has fixed a bin problem. No action is required. This notice will be logged in the event and error buffers.

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 HT_SIZE in acfg.h.

SMXE_INIT_MOD_FAIL Occurs when the call to smx_modules_init() in the Protosystem fails. This routine is called by ainit() to initialize the smx component modules (e.g. smxFS, smxUSBH, etc.). If this error occurs, step through this routine to determine which initialization routine failed.

SMXE_INSUFF_DAR Not enough space in the specified DAR to get a block of the requested size. More space needs to be allocated for the DAR. See discussion in Dynamically Allocated Regions section in smx_Base manual showing how to allocate DAR memory.

SMXE_INSUFF_HEAP Not enough heap to allocate a block of the requested size. More space needs to be allocated for the heap. One way to do this is to increase HEAP_SPACE in acfg.h. During operation, other methods can be used, such as calling smx_HeapRecover().

SMXE_INSUFF_UNLOCKS Reported by smx_TaskLockClear() if smx_lockctr is not 1, as expected. Indicates that the number of unlocks does not match the number of locks due to a programming error or unexpected execution path.

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_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_PARM An invalid parameter, not covered by other error types, has been passed to an smx call. Check parameters vs. description in RM for the smx service.

SMXE_INV_PCB Pool handle does not point to a valid pool control block Check if smx_BlockPoolCreate() or smx_BlockPoolCreateDAR() 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_QCB Queue handle does not point to a valid queue control block Check the eq parameter of smx_EventQueueSignal().

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 (2exp31 - 1) need be added to etime to convert to a tick timeout.

SMXE_INV_TMCB 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 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_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() due to a programming error or unexpected execution path.

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(), if necessary in special situations.

SMXE_NO_CBLKS This error results from an smx create call, such as smx_TaskCreate(), if it is unable to allocate a needed control block pool from SDAR. (Create calls automatically allocate control block pools the first time they are called.) This error indicates that there is insufficient SDAR.

SMXE_NULL_PTR_REF The value at address 0 changed since initialization, which suggests a null pointer was used. This is checked in the idle task and at exit. See main.c. This checking is enabled by NULL_PTR_REF_CHECK in acfg.h. It should only be enabled for targets that have RAM at 0, since it is useless if flash is at address 0, and could cause a processor fault if no memory is at 0. It may be that RAM is mapped to 0 only for some build targets such as Debug but not others, in which case a check of SMX_BT_DEBUG or similar could be added to conditionally enable it.

SMXE_OBJ_IN_USE smx++ error. Occurs when a destructor has been called and the object is still in use. See smx++ manual.

SMXE_OBJ_NOT_CREATED smx++ error. Occurs when an object method is used and the object has not been created. See smx++ manual.

SMXE_OK No system error.

SMXE_OP_NOT_ALLOWED Occurs when a limited SSR is called from an LSR. Programming error.

SMXE_OUT_OF_BCBS, SMXE_OUT_OF_MCBS, SMXE_OUT_OF_MUCBS, SMXE_OUT_OF_PCBS, SMXE_OUT_OF_PICBS, SMXE_OUT_OF_QCBS, SMXE_OUT_OF_TCBS, SMXE_OUT_OF_TMCBS

Out of control blocks of the type specified. This type of error occurs when a create call is unable to get a control block with smx_ControlBlocksGet_F(). 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 NUM_TASKS in acfg.h should be increased. However, it can also mean that SDAR is too small and no pool was created. SDAR is sized automatically in APP\mem.c based upon configuration settings, but alignment requirements of CB pools may require additional padding. In this case, increase SDAR_SIZE_ADD in acfg.h.

SMXE_OUT_OF_STKS Applies to shared stacks from the stack pool. If stack scanning is not enabled, out of stacks occurs if the freestack pool is empty, when a task that is being dispatched needs a stack (i.e. it is *unbound*). If stack scanning is enabled, it occurs if both the freestack and scanstack pools are empty, when a task that is being dispatched needs a stack. 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. 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. To the extent that the resulting performance is acceptable, it is possible to have fewer than the number of shared stacks that might be needed at one time. Otherwise, it is necessary to increase the value of NUM_STACKS in acfg.h.

SMXE_Q_FIXED The scheduler detected a broken link in the ready queue and was able to fix it.

SMXE_RQ_ERROR Detected by the scheduler during task dispatch. The scheduler will attempt to fix rq.

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 (tcb.flags.stk_chk set). Indicates that the task's stack pointer exceeds the stack top (tcb.stp) or that the stack high water mark (tcb.shwm) exceeds the stack size (tcb.ssz). If a stack pad is present and nothing else has been damaged, operation continues normally. Otherwise: if the stack block has been exceeded (i.e. memory "above" it has been damaged), smx_EMExitHook() is called, which the user can implement to stop the task, abort the application, etc. Stack pad size is set by STACK_PAD_SIZE in acfg.h. One size applies to all stacks, except that SS has no stack pad. Ample sizes are recommended during debugging. Another possible cause of this error is use of a foreign stack when a task switch occurs.

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. See also UG Stacks.

SMXE_TMR_STOPPED A timer operation cannot be performed because the timer has already stopped.

SMXE_UNKOWN 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 timeout parameter for SSRs that can wait.

SMXE_WRONG_MODE A create function has an unrecognized mode.

SMXE_WRONG_TYPE_QUEUE Indicates an attempt to access a wrong type queue — for example receive a message from a semaphore, send a signal to an exchange, etc.

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 See UG, Service Routines, interrupt service routines.

SMX_PRI_NOCHG No change to task's priority. Used in smx_TaskBump() and smx_TaskStartNew(). Defined in xdef.h.

SMX_TMO A timeout has occurred for the last smx service from the current task.

SMX_TMO_INF Infinite timeout. Used in SSRs that have a timeout parameter. Sets the task's timeout to 0xFFFFFFFF, which is the timeout disabled value.

SMX_TMO_NOCHG No change to task's timeout. This timeout value only makes sense in an SSR that is acting upon another task, such as smx_TaskStop() or smx_TaskSuspend(). See the discussions of these services for more details. If this timeout value is used in an SSR that acts upon the current task (e.g. smx_MsgReceive()), the result is the same as INF.

SMX_TMO_NOWAIT No timeout — i.e. do not wait. Use of this value for a timeout parameter 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, especially free software. Such software frequently uses the heap and may use it badly.

srnest **service routine nesting** level, stored in smx_srnest. Records the nesting level of service routines. When any smx ISR, LSR, or SSR starts, srnest is incremented, and when it finishes srnest is decremented, unless it is 1, in which case the scheduler may be started. For example, an LSR may call an SSR, which is interrupted by an ISR. At this point srnest would be 3. (Note that for ARM-M processors, srnest is not incremented/decremented in ISRs because a hardware mechanism is used, instead.)

SS See **system stack**.

SSR **system service routine**. Most smx services are implemented as SSRs. An SSR is a function which starts with smx_SSR_ENTER() and ends with smx_SSR_EXIT(). Between the enter and exit macros, operations can safely be performed on smx objects, such as control blocks. SSRs are task-safe. An SSR can call another SSR, in which case, the programmer must ensure that there will be no smx object conflicts. See UG Service Routines and RM smx_SSR_ENTER() and smx_SSR_EXIT() for discussion about writing SSRs.

stack Each task requires a stack when it is running or suspended (but not when it is stopped). Tasks can have either permanent stacks or shared stacks. A permanent stack remains bound to a task as long as the task is not deleted. A shared stack is bound to a task when the task is dispatched and returned to the stack pool when the task is stopped. smx allows tasks to be stopped while waiting for events such as semaphore signals, messages, etc. Such tasks are called one-shot tasks.

Permanent stacks are allocated from the heap. The size of each stack may vary for each task and is set by the stack size parameter in smx_TaskCreate(). Shared stacks are allocated from the stack pool which is created by smx_Go(). The size of stack pool stacks is set by STACK_SIZE in acfg.h. See UG Stacks.

stack high water mark Actual stack usage is stored in tcb.shwm for easy inspection in the debugger or via smxAware. The stack high-water mark indicates the maximum stack usage by the task, even if the it does not have a permanent stack. It is used to detect overflow and to tune stack sizes.

stack pool The stack pool is allocated from ADAR by smx_AllocStacks(), the first time that smx_TaskCreate() is called. Stacks are also filled with the scan pattern. Stacks in the stack pool are shared between one-shot tasks. NUM_STACKS, STACK_SIZE, and STACK_PAD_SIZE, in acfg.h control stack pool attributes.

stack scan Permanent stacks are filled with a known pattern when bound to tasks. Shared stacks are filled when put into the freestack pool. Each stack is periodically scanned by the idle task, from the top of the stack (pad) down to the first change of pattern. The difference between tcb.sbp (stack bottom pointer) and this is recorded as the stack high water mark in tcb.shwm. Stack scanning is a very reliable means to determine maximum stack usage, unless the stack overflows past the pad. In this case, make the pad bigger during development.

stack size Task stack size is determined by the stack size parameter in smx_TaskCreate() when a task is created or by STACK_SIZE in acfg.h for stack pool stacks. Stacks must be large enough for the maximum nesting of functions and SSRs called by the task. Actual stack size may be a little less than requested, due to alignment on an SB_STACK_ALIGN boundary. Space allocated from the heap == stack size + RSA_SIZE + STACK_PAD_SIZE. The size for stack pool stacks is the same, except alignment bytes are added. See also UG Stacks.

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 to 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, run C++ static initializers, and then branch to main(). See QS: smx Startup and Scheduler Operation.

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 TCB 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 at a time may be in the **run** state. 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.

static block is a data block which is statically defined, e.g.:

u8 block[100];

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 DAR or the heap.

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, it is necessary that all smx create calls automatically create any smx objects that they need, if not already created. For example, the first smx_TaskCreate() call initializes SDAR, if not already done, then allocates the TCB and stack pools.

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. See UG etime and stime and SB sb_StimeSet.

stop ends execution of a task such that it cannot be resumed and must be restarted from its beginning. When the current task stops itself with a stop call, it is dequeued from rq, put into the wait state, and is usually enqueued on a wait queue for an event. When ct stops another task with smx_TaskStop() or smx_TaskStart(), the other task is dequeued from any queue it may be in. Either way, a stopped task returns its stack to the stack pool, unless its stk_perm flag is set, and its stack pointer is cleared and thus its context is lost. If a test condition (e.g. semaphore set) is satisfied immediately, task stop still occurs, but it is followed immediately by task start.

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 SSRs.

stop on To stop a task on a queue means to stop the task, then enqueue it on the queue.

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 the current task suspends itself, with a suspend SSR, it is dequeued from rq, its context is saved in its RSA, and it usually is enqueued on a wait queue for an event. When ct suspends another task, using smx_TaskSuspend() or smx_TaskResume() the other task is dequeued from any queue it may be in. Either way, the suspended task retains its stack and its stack pointer is saved in its TCB. If a test condition (e.g. semaphore set) is satisfied immediately, the task simply continues.

suspend call An SSR that causes a task to be suspended, unless the expected condition is satisfied immediately.

suspend on To suspend a task on a queue means to suspend the task, then enqueue it on the queue.

system service A service provided by smx. It can be an SSR, bare function, or a macro. Only SSRs are task-safe. All smx system services are prefixed with “smx_”. smxBase services may also be referred to as system services. These are limited to bare functions or macros and are prefixed with “sb_”. See SB Base and Utility.

system stack The system stack (**SS**) is used for startup, initialization (including C++ static initializers), ISRs, LSRs, the scheduler, 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. It is recommended that SS be located in on-chip SRAM for best performance of ISRs, LSRs, and the smx scheduler.

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(). See UG Tasks and UG One-Shot Tasks.

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 UG Service Routines.

task state — See **state**.

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 hooked exit and entry routines.

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 switch — occurs due to preempting, stopping, or suspending the current task and starting or resuming another. Performed by the smx task scheduler.

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. See UG Tasks and UG One-Shot Tasks.

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 — **TCB pointer** type. Pointers of this type store task handles.

tc — See **top chunk.**

temporary stack — A stack given to a one-shot task from the stack pool when it is dispatched. The stack is released back to the stack pool when the task stops.

TG — **SMX Target Guide** — abbreviation used in citations.

thrashing — Means that many unnecessary task switches are occurring. These waste processor bandwidth and can hurt performance. smx provides mechanisms, such as task locking, to control thrashing.

thread — Short for “thread of execution.” In the broad definition, as used for smx, includes tasks, LSRs, and ISRs.

thread-safe — Means the same as task-safe.

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 UG Semaphores, threshold semaphore.

TickISRHook() Callback function to hook into the tick ISR. Can be used to piggyback ISRs on the tick interrupt for testing or to add more capability to the tick ISR without modifying it.

tight heap A heap that has very little margin because of insufficient RAM and thus is prone to failure due to fragmentation.

timeout All calls which put tasks into the wait state permit a timeout to be specified. Timeouts ensure that tasks will not wait forever, and they also break task deadlocks. SMX_TMO_NOWAIT can be specified if no wait is desired. SMX_TMO_INF can be specified if no timeout is desired. A seldom-used parameter, 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 (2exp31-1) ticks, which is the maximum value of etime. The resolution of task timeouts is controlled by TIMEOUT_PERIOD in acfg.h.

timeout[] smx_timeout[] is an array of 32-bit timeouts, one per TCB. Task timeouts (not to be confused with "timers") are in the same order as TCBs in the TCB pool. 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 (smx_tmo_min) is periodically compared to smx_etime. This comparison is performed 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 it into smx_tmo_min. If it is also 0 (i.e. two tasks have timed out at once), smx_TimeoutLSR() invokes itself, which allows other LSRs to run before it runs again. This timeout mechanism achieves low overhead, if the number of tasks is not too great and/or the resolution is not too fine. See also UG Timing, timeouts.

TIMEOUT_PERIOD is a configuration setting in acfg.h that sets smx_cf.tmo_period, which controls how often smx_tmo_min is compared to smx_etime — i.e. how often smx_TimeoutLSR() runs. Hence, it controls the resolution of task timeouts. Small resolutions, such as 1 tick, are used for accurate timing. Large resolutions such as 10 or 100 ticks are used for safety.

timer A timer is a system object, consisting of a timer control block (TMCB) linked into the timer queue (tq). smx supports both one-shot and cyclic timers. Both are created and started by smx_TimerStart(), which allows specifying a relative time from now and a cycle time. If the cycle time is zero, the timer is called a one-shot timer; it is automatically deleted after it times out. Otherwise, the timer becomes a cyclic timer with the specified cycle time. Cyclic timers are requeued immediately so there is no cumulative timing error.

Timers are enqueued in tq in order of their times, each with a calculated differential time in its TMCB. Decrementing of the first TMCB counter is done by smx_KeepTimeLSR(). Operation is similar to an event queue. Timers operate with a 1 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.

time-slice scheduling is a task scheduling algorithm in which each task is given an equal time to run. smx provides time slicing only for priority 0 tasks. SMX_CFG_TIME_SLICE in acfg.h is the time slice period. If 0, time slicing is disabled. Note that preempting tasks will use time from a time-sliced task's period. See also preemptive scheduling and round-robin scheduling.

TMCB **timer control block** A timer control block is assigned to a timer and initialized when the timer is started by smx_TimerStart(). A TMCB has forward and backward links to link into the timer queue, owner, interval, differential count, LSR, par, and a handle pointer. These are used by timer services and the smx_TimeoutLSR(). See UG Timers.

TMCB pool All TMCBs are in a pool, which is controlled by the smx_tmcbs pool control block. The singly-linked list of free TMCBs is pointed to by smx_tmcbs.pn. The next-link pointer is in the first word of each free TMCB. The last free TMCB has a NULL link.

TMCB_PTR **TMCB pointer** type. Pointers of this type store timer handles.

token is an object, such as an smx message, that represents a resource. Whatever task possesses the token is allowed to access the resource. This can work better than a resource semaphore or a mutex, because a token can contain information about the resource that enables the task to use it.

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 in the heap. Initially, it and the donor chunk contain all free heap space. Allocations which cannot be satisfied by the SBA, donor chunk, nor larger bins come from tc. It must be large enough to hold a chunk control block (CCB).

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 the current task. (If ct is locked, some other task could be current task.)

tq **timer queue**, is pointed to by smx_tq. See also: timer.

thread same as task.

TRUE 1. (However, test for !0.)

u8 unsigned 8-bit integer.

u16 unsigned 16-bit integer.

u32 unsigned 32-bit integer.

UBA See **upper bin array.**

UG **smx Users Guide** — abbreviation used in citations. Usually followed by a Chapter Name or section name.

unbound mode Task has no stack. See **bound mode**.

unbounded priority inversion See **priority inversion**.

unlocked See **locked**

upper bin array (UBA) That portion of smx_bin[] that is above the SBA. It contains *upper bins*.

unrestricted macro can be invoked from any service routine or task.

use_dc mode is controlled by the **smx_heap.mode.use_dc** flag. It starts in the ON state. It can be set ON or OFF by smx_HeapSet(). When ON, an SBA-size chunk is taken from the donor chunk if the SBA bin for it is empty. When OFF, allocation skips dc and goes to the next larger occupied bin, instead. If no space is allocated for the donor chunk (HEAP_DC_SIZE is 0 in acfg.h) then use_dc should be turned OFF. It can also be turned off to improve performance, if dc becomes too small.

volatile registers The registers that the C/C++ compiler expects to be changed by a function call, and therefore saves them before the function call and restores them after. See also non-volatile registers.

XCB **exchange control block** Each exchange has an XCB, which contains important exchange parameters. These include forward and backward links for task and message queues, mode, task queue flag, message queue flag, and name. See UG Exchange Messaging.

XCB pool See QCB pool.

XCB_PTR XCB pointer type. Variables of this type contain exchange handles.