smx++

Developer's Guide

Version 4.4

July 2020

by
Alan Moore
and
Ralph Moore

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

Revisions

date	ver	comments
4/93	3.0	first release
5/96	3.3	preliminary v3.3. Major revision including documentation of new features
2/98	3.3	final v3.3. Additional changes.
7/01	3.5	update for v3.5
5/04	3.6	update for v3.6
5/05	3.7.0	update for v3.7
11/05	3.7.1	changes to smx_Task class, addition of smx_Mutex, and minor changes
11/10	4.0	update for v4.0
4/13	4.1.1	update for v4.1.1
4/14	4.2.1	update for v4.2.1
5/15	4.3	update for v4.3.0
7/20	4.4	update for v4.4.0

Table of Contents

Contents

Introduction

Overview

smx++ is an abstraction layer over smx, which allows C++ programmers to create complex application software with minimal understanding of the underlying smx RTOS kernel. It supports object-oriented programming and provides a simpler smx API for C++ programmers. smx++ offers the following classes:

smx_Object
smx_Block
smx_EventGroup
smx_EventQueue
smx_Msg
smx_MsgXchg
smx_Mutex
smx_Pipe
smx_Pool
smx_Sem
smx_Task
smx_Timer

These are meant to serve as base classes for derived application classes and for member objects of application classes. Each class, except smx_Object offers a full set of methods corresponding to underlying smx services.

smx++ fosters dual-language implementations where C++ and C programmers can work together on the same project. It is thought that smx++ will be used primarily for data processing, user interfaces, communication with the Cloud, and other high-level software, which benefits from object-oriented design. It is expected that low-level functions such as ISRs, LSRs, BSP code, and basic tasks will be written in C or assembly. However, nothing prevents using C++ classes and methods for low-level operations, if preferred.

smx++ is augmented by the very fast smx heap and by stack sharing between one-shot tasks. These make complex object-oriented applications practical on modest embedded systems with limited memory and processor power. For more discussion of this, see Appendix A.

smx++ is compatible with Embedded C++ and has been developed with the same goals of speed and austere memory usage in mind. smx++ is intended to provide the power and modularity of object oriented programming while being small and fast enough to work effectively in a small embedded environment.

About This Manual

Novice C++ programmers who are experienced C programmers will find smx++ helpful to learn C++ programming. Only a basic knowledge of C++ is necessary to use smx+. It does not employ advanced C++ features. This manual is written to help the novice learn to use C++ and thus determine if it is useful for his or her project. Reading class and method descriptions and studying the examples should help to accomplish this. Also, reading Appendix A is important to understand how memory space for objects is allocated.

Most smx++ methods are thin wrappers around smx services. In the interest of making smx++ easy to use, method descriptions are intentionally brief and do not specify all errors and side-effects that may occur. If this information is needed, refer to the smx Reference Manual for the underlying smx service.

It is recommended that the smx User's Guide be read, at least in part. Many concepts explained therein carry over to smx++.

General Concepts

smx++ Classes

smx++ is comprised of 11 classes that are derived from the smx_Object class. The primary purpose of the smx_Object class is to provide space for smx++ objects. Since application objects will be derived from smx++ objects, it also provides space for them. In the smx_Object class, the *new* operator is overridden with sb_BlockGet(), which gets a block from the *Global Pool*. This pool is automatically created from the user configuration constants PP_OBJ_SIZE and PP_OBJ_NUM in acfg.h. A size of 8 bytes is adequate for all smx++ objects.

If a derived object requires a larger block, smx_HeapMalloc() is called to get the block from the smx heap. As compared to a standard C compiler heap, the smx heap is thread-safe and offers many features intended for embedded systems. It also is very fast. smx_Object also provides a *delete* operator to release blocks back to GlobalPool or to the heap, when objects are destroyed.

Because smx++ classes and methods are based upon smx, they may not be used in ISRs. However, they can be used in LSRs, provided that no wait times are specified.

smx Control Blocks

Each smx++ object contains a protected data member which links to the underlying smx control block. Information in the smx control block is accessed via this link and is not repeated in the smx++ object. Hence, fields in a control block are technically protected members of the smx++ object.

It should not be necessary to access smx control blocks directly. Peek() and other methods are provided to get necessary information. It is important not to access smx control blocks directly for the following reasons:

(1) Isolation from the internal workings of smx.

(2) Independence from the structure of smx control blocks.

(3) Added layer of security by preventing unintentional modification of critical data.

If you need to access a control block directly, you should create your own methods in a derived class to access what you need. For example:

```
class MyTask : public smx_Task
{
      CB_PTR GetForwardLink(void)   { return TaskCBP->fl; }
      ...
}
```

Inline Methods

Since most smx++ methods are small, they are declared to be *inline*. This generally results in faster operation and reduces code space if the overhead for a function call exceeds the size of the method. These methods are defined in .hpp header files after the class definition. Larger methods are put in corresponding .cpp files.

Error Handling

smx++ error handling is an extension of smx error handling and uses the smx error manager, smx_EM in xem.c. Since most methods utilize underlying smx functions, most errors for smx++ are actually smx errors. Some new errors have been created for smx++ — see Appendix A for a list. smx++ errors are identified as smx errors — e.g. SMXE_NOT_HOLDING. See xdef.h for the list of all smx and smx++ errors.

Constructor Errors

Most smx++ constructors attempt to get an smx control block. If this should fail, the instance is still constructed, but the control block pointer is set to NULL. For most classes, this is an indication to its methods that the smx object could not be created. Such methods will abort, report SMXE_OBJ_NOT_CREATED, and return 0 or FALSE. Normally an smx or smxBase error message will precede, explaining the reason for the object not being created — e.g. SMXE_OUT_OF_TCBS.

Objects of the smx_Msg and smx_Timer classes can be either *holding* or *non-holding* meaning that they have an underlying smx control block or not. The methods of these classes deal with both cases.

Task Queues

All task queues are prioritized. The *first* task in a task queue is the highest priority and longest-waiting task at that priority.

smx++ Class Reference

This section is provided as a reference to all smx++ classes, methods, and data members. It is organized alphabetically by class name.

Format of the Class Descriptions

file Indicates the name of the C++ file containing the source code for the class.

#include The name of the C++ include file containing the definition of the class structure.

derived from If the class has been derived from another smx++ class then this is the name of that class.

Following the "derived from:" field is a brief description of the class, its uses, things to watch out for, etc.

Example These are intended to demonstrate how the class might be used in an embedded application.

Data Members Description of all public and protected data members and how they can be used.

Constructors and Destructor Declarations for all constructors and destructor.

Methods Declarations for all public and protected methods.

Following the Methods field are sections describing each of the constructors followed by descriptions of each method in the class.

Format of Methods Sections

First is the prototype declaration for the method complete with return and parameter types. For example: smx_EventQueue::smx_EventQueue(const char *name).

A brief synopsis of the method is given along with important notes, warnings, and things to look out for. This is followed by several fields:

smx functions called Indicates what smx functions may be called by this method. It is often useful to look up these functions in the smx Reference Manual to learn more about what the method is doing and what errors it might report. Many methods are wrappers for standard smx calls.

Returns Describes possible values returned by the method, if any. Generally, a return value of 0, FALSE, or NULL indicates that the method failed for some reason, possibly due to an error condition or timeout.

par	Stop methods do not return a value. Instead they pass value as the parameter of the task main function, when it restarts.
Errors	If present, lists smx++ errors. Errors for smx functions called are not listed. Please see the smx Reference Manual call descriptions for those.
Example	These briefly illustrate the common uses of methods.

Terminology

Object	References to “block”, “event queue” mean an object of that type. The word “object” is generally omitted.
Bare function	is an ordinary function. The term *bare* is used to emphasize that it is not an smx service routine (SSR). Hence, it is not preemption-safe and should not be used from tasks unless otherwise protected (e.g. disabling interrupts).

smx_Object Class

file xsmx.cpp

#include xsmx.hpp

derived from none

smx_Object class is a pure virtual base class used for all smx++ objects.

Public Data Member

static PCB_PTR * **GlobalPool**;

Since it is static, this is effectively a global variable which is used by all derived classes.

Constructors and Destructor

None

Operators & Methods

void operator **delete**(void * ptr);
void * operator **new**(size_t size);
u32 **EtimeGet**(void);
u32 **StimeGet**(void);
virtual SPP_CLTYPE **WhatIs**(void); // returns SPP_CL_OBJ

void * operator **smx_Object::new**(size_t size)

This new() operator is used to allocate objects for smx++ and its derived classes. If size is less than or equal to PP_OBJ_SIZE in acfg.h, then the object is allocated from the GlobalPool. Otherwise, the object is allocated from the smx heap. The object is also allocated from the smx heap if GlobalPool is empty.

GlobalPool is automatically initialized and its base pool is created the first time an smx++ or an smx++ derived object is constructed. User configuration constants PP_OBJ_NUM and PP_OBJ_SIZE in acfg.h control the size of the pool and the size of the blocks that it contains. smx++ objects require only 8 bytes. However, the minimum heap block is 16 bytes, so PP_OBJ_SIZE is currently set to 12. This allows for somewhat larger derived objects, before going to the heap.

The new smx heap is fast, so it should work well for smx++ derived objects. If not, smx_Object::new() can be modified to use multiple base pools. If you prefer to use smx heap for all objects, set PP_OBJ_NUM and PP_OBJ_SIZE to 0 in acfg.h or modify smx_Object::new().

smx functions called sb_BlockGet(), smx_HeapMalloc(), and sb_BlockPoolCreateDAR(),

Returns	block pointer	if block allocated
	0	block not available.

Errors none

Example smx_Task *TaskAP = new smx_Task(HI, 200, SMX_FL_NONE, "TaskA");

Constructs TaskA pointed to by TaskAP. TaskA is an object of the smx_Task class. Here the smx_Object new operator overrides the new operator provided by the compiler.

void operator **smx_Object::delete**(void * ptr)

This delete() is used to free objects allocated with the smx_Object::new operator. If ptr points within GlobalPool, it calls sb_BlockRel() to release the block back to GlobalPool. Otherwise, it calls smx_HeapFree() to release the block back to the heap.

smx functions called sb_BlockRel() or smx_HeapFree()

Returns none

Errors none

Example delete (TaskAP);

Deletes the TaskA object created above. Here the smx_Object delete operator overrides the delete operator provided by the compiler.

u32 **smx_Object::EtimeGet**(void)

Returns etime, in ticks. See smx Users Guide section on Timing, etime, and stime. This provides a method to get etime within any smx++ or smx++ derived object.

smx functions called none

Returns smx_etime

Errors none

Example

u32 time;

time = EtimeGet();

u32 **smx_Object::StimeGet**(void)

Returns stime, in seconds. See smx Users Guide section on Timing, etime, and stime. This provides a method to get stime within any smx++ or smx++ derived object.

smx functions called none

Returns smx_stime

Errors none

Example u32 time = StimeGet();

smx_Block Class

file xblk.cpp

#include xblk.hpp

derived from smx_Object class

The smx_Block class gets its memory from the heap or from a DAR. Blocks are discussed in more detail in the smx Reference Manual. The smx_Block class provides a convenient C++ interface to the smx BCB.

Protected Data Members

u32 **ObjectFlags**;
BCB_PTR **BlockCBP**;

Constructors and Destructor

smx_Block(smx_Pool &pool, u32 clrsz = 0);
smx_Block(u8 *bp);
smx_Block(BCB_PTR p);
virtual ~**smx_Block**();

Methods

u32 **Peek**(SMX_PK_PARM par);
virtual SPP_CLTYPE **WhatIs**(void); // returns SPP_CL_BLK

smx_Block::smx_Block(smx_Pool &pool, u32 clrsz)

The constructor will get a block from the pool and clear clrsz bytes.

smx functions called smx_BlockGet()

Returns none

Example

```
smx_Block * bp  = new smx_Block( pool );
```

This example gets a block from *pool* and returns a pointer to it. Since clrsz is not specified, it defaults to 0 and no bytes will be cleared.

smx_Block::smx_Block(u8 *bp)

This constructor takes a pointer to an existing block of memory and creates an smx_Block with it.

smx functions called smx_BlockMake()

Returns none

Example
```
u8 * bp = smx_HeapMalloc( 100 );
smx_Block * blkp = new smx_Block( bp );
```

Get 100 bytes from the heap and create a block using it. blkp points at the block.

smx_Block::smx_Block(BCB_PTR p)

The conversion constructor creates an smx_Block from an existing smx block.

smx functions called none

Returns none

Example
```
PCB_PTR  pool;
…
BCB_PTR p = smx_BlockGet( poolA, NULL, 0 );
smx_Block * blkp  = new smx_Block( p );
```

Get an smx block from poolA and construct a block from it. blkp points at the block.

smx_Block::~smx_Block()

The destructor will release the data block back to its base pool, if it came from a base pool. Also releases the BlockCBP's BCB back to its pool.

smx functions called smx_BlockRel()

Returns none

u32 **smx_Block::Peek()**(SMX_PK_PARM par)

Returns the value of the specified parameter for this block. See smx_BlockPeek() in the smx Reference Manual for a list of block parameters.

smx functions called smx_BlockPeek()

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

Example

```
smx_Block * blkp = new smx_Block( pool );
u8 *bp = (u8*)blkp->Peek( SMX_PK_BP );
*bp = 0x55;
```

In this example, we obtain a pointer, bp, to the data for the block, blk, and then load 0x55 into the first byte of the data area.

.

smx_EventGroup Class

file xeg.cpp

#include xeg.hpp

derived from smx_Object class

The smx_EventGroup class is used to allow tasks to wait on a specific event or logical combination of events. Please refer to the smx User's Guide for more information on smx Event Groups.

Protected Data Members

u32 **ObjectFlags**;
EGCB_PTR **EventFP**;

Constructors and Destructor

smx_EventGroup(u32 mask = 0, const char *name = NULL);
virtual **~smx_EventGroup**();

Methods

BOOLEAN **Clear**(u16 init_mask = 0);
u32 **Peek**(SMX_PK_PARM par);
BOOLEAN **FlagsPulse**(u16 set_mask);
BOOLEAN **FlagsSet**(u16 set_mask, u16 clear_mask = 0);
u32 **FlagsTest**(u32 test_mask, u16 clear_mask = 0, u32 timeout = INF);
void **FlagsTestStop**(u32 test_mask, u16 clear_mask = 0, u32 timeout = INF);
virtual SPP_CLTYPE **WhatIs**(void); // returns SPP_CL_EG

Example

```
#define SIGPWR  0x01

class PowerMgr : public smx_Task
{
    public:
        PowerMgr() : smx_Task(PRI_MAX, 0, SMX_FL_NONE, name) { }
        ~PowerMgr()   {  if (eg) delete eg;  }

        void SigPwr()  {  eg->FlagsSet(SIGPWR);  }

        void Main(u32 par)
        {
            eg = new smx_EventGroup();
            while (eg->FlagsTest(SIGPWR, 0))
            {
                // Handle power loss.
            }
        }
    private:
        smx_EventGroup *eg;
};

PowerMgr  PowerLossTask();
```

This example simulates a Linux-style signal to a task. The idea here is to create a task with an integrated event group that, when receiving a SIGPWR power-loss signal, will wake up and handle the power loss event. This is achieved by deriving the PowerMgr class from smx_Task and including a private smx_EventGroup member.

PowerLossTask is created as a top priority task so that it will preempt any normal task and deal with the power loss, immediately. Its Main() method creates the eg object and then waits for the SIGPWR flag to be set. The public SigPwr() method provides a means for another task to set the SIGPWR flag in eg, as follows:

```
PowerLossTask.SigPwr;
```

which wakes up PowerLossTask. Other flags could also be declared and tested in this object in addition to the SIGPWR flag.

Note: PowerLossTask will be created in compiler startup code before C main() is called. That means that the PowerMgr constructor will be called before the smx environment has been initialized. To deal with this, smx objects have been designed to be dynamically self-initializing. Hence, when the first smx_Task constructor calls smx_TaskCreate(), it will create the TCB pool and get a TCB, create the stack pool so the scheduler can allocate a stack (note that stack size == 0, above), and intialize PowerLossTask.

smx_EventGroup::smx_EventGroup(u32 init_mask = 0 , const char *name = NULL)

Constructs an event group with its flags set to init_mask.

smx functions called smx_EventGroupCreate()

Returns none

Example See smx_EventGroup class example

smx_EventGroup::~smx_EventGroup()

Resumes all waiting tasks with FALSE return values, deletes the underlying smx event group, then destroys this event group.

smx functions called smx_EventGroupDelete()

Returns none

Example smx_EventGroup class example

BOOLEAN **smx_EventGroup::Clear**(u16 init_mask = 0)

Resumes all waiting tasks with 0 return values and sets the event group flags to init_mask.

smx functions called smx_EventGroupClear()

Returns

TRUE	event group cleared
FALSE	event group not cleared due to an error

Example

```
#define  TXRDY  0x60
smx_EventGroup  Modem; // global object with flags cleared

Xmit::Stop()
{
      Modem.Clear( TXRDY );
}
```

This example clears Modem and sets its transmit ready flag.

u32 **smx_EventGroup::Peek**(SMX_PK_PARM par)

Returns the value of the specified parameter for this event group. See smx_EventGroupPeek() in the smx Reference Manual for a list of event group parameters.

smx functions called smx_EventGroupPeek()

Returns

value	Value for par.
0	Value, unless error.

Example

```
smx_EventGroup *egp;
u32 num_tasks;
TCB_PTR first_task;

ModemP = new smx_EventGroup();
...
num_tasks = egp->Peek(SMX_PK_TASK);
if (num_tasks > 0)
    first_task = (TCB_PTR)egp->Peek(SMX_PK_FIRST);
else
    first_task= NULL;
```

BOOLEAN **smx_EventGroup::FlagsPulse**(u16 set_mask)

Like FlagsSet() except that it leave only flags set that were already set. Note: if any tasks are resumed, some initially set flags may be reset — see FlagsTest().

smx functions called smx_EventGroupPulse()

Returns TRUE flags pulsed
FALSE flags not pulsed due to error.

Example

```
#define F2 0x2
#define F1 0x1
smx_EventGroup eg(F1);  // global eg with F1 set.

void taskA::Main( u32 par )
{
    ...
    eg.FlagsPulse(F2);
}
```

Resumes tasks waiting for F2 or for F2&F1, since F1 is already set. F2 is not left set, but F1 is left set (unless a resumed task resets it).

BOOLEAN **smx_EventGroup::FlagsSet**(u16 set_mask, u16 pre_clear_mask = 0)

Clears flags selected by 1's in pre_clear_mask, then sets flags selected by 1's in set_mask. Then resumes tasks with flag matches. See smx_EventFlagsSet in the smx Reference Manual for details.

smx functions called smx_EventGroupSet()

Returns TRUE flags changed
FALSE flags not changed

Example

```
smx_EventGroup *ModemP; // global pointer

Sys_Init()
{
    ...
    ModemP = new smx_EventGroup();
}

Xmit::Start()
{
    ...
    ModemP->FlagsSet( TXRDY, RXRDY );
}
```

This example clears the RXRDY flag, then sets the TXRDY flag. These are mutually exclusive flags, which must not be simultaneously true. Note that ModemP is constructed during initialization and is globally accessible via its global pointer.

u32 **smx_EventGroup::FlagsTest**(u32 test_mask, u16 post_clear_mask = 0, u32 timeout = INF)

Tests for a match between event group flags and test_mask. If found, the current task is continued, and the flags causing the match are returned. Also clears event group flags selected by the post_clear_mask. Suspends current task if no match is found and timeout > 0. If a match does not occur before the timeout occurs, the task is resumed with a 0 return. See smx_EventFlagsTest in the smx Reference Manual for details.

smx functions called smx_EventFlagsTest()

Returns

flags	flags causing match
0	no match, timeout, or error

Example See smx_EventGroup class example

void **smx_EventGroup::FlagsTestStop**(u32 test_mask, u16 post_clear_mask = 0, u32 timeout = INF)

This method works like FlagsTest(), but it first stops the current task. As soon as the proper flag condition is met the task is restarted.

smx functions called smx_EventFlagsTestStop()

Returns none

par

flags	flags causing match
0	no match, timeout, or error.

Example

```
#define  AND     0x8000
#define  TXRDY 0x60
#define  DSR     0x20
#define  CTS     0x10

smx_EventGroup * ModemP;

Sys_Init()
{
      ...
      ModemP = new smx_EventGroup();
}

Transmit::Main( u32 par )
{
      // prepare and send message
      ModemP->FlagsTestStop( AND | TXRDY | DSR | CTS );
}
```

In this example, the Task is stopped and then restarted when all flags are set. Every time the Transmit task is started it prepares and sends a new message.

smx_EventQueue Class

file xeq.cpp

#include xeq.hpp

derived from smx_Object class

The smx_EventQueue class is used to count the number of times an event occurs. This is most commonly used as a delay mechanism for tasks. See the "Event Queues" chapter in the smx Users Guide for more information.

Protected Data Members

u32 **ObjectFlags**;
EQCB_PTR **EventQueueCBP**;

Constructors and Destructor

smx_EventQueue(const char *name = NULL);
virtual **~smx_EventQueue**();

Methods

BOOLEAN **Count**(u32 count, u32 timeout = INF);
void **CountStop**(u32 count, u32 timeout = INF);
BOOLEAN **Clear**();
BOOLEAN **Signal**();
virtual SPP_CLTYPE **WhatIs**(void); // returns SPP_CL_EQ

Example

```
class EventDemo : public smx_Task
{
public:
	EventDemo(const char *name = NULL) :
		smx_Task(PRI_NORM, 0, SMX_FL_NONE, name), counter(0)  {}
	virtual void Main( u32 par );
	void Signal()   {  ec.Signal();  }
protected:
	smx_EventQueue ec;
	u32 counter;
};

EventDemo     ECT("EventDemo");

void EventDemo::Main(u32 par)
{
	while (1)
	{
		ec.Count(2);
		counter++;
	 }
}
```

In this example the technique of combining a task object with an integrated smx_EventQueue object is demonstrated. The EventDemo class is derived from smx_Task so that it inherits all of the capabilities of a task. For simplicity, the class EventDemo is created with a simplified constructor. It has an integrated Main() method which will wait for 2 signals to it's event counter. The Signal() method is brought out to the EventDemo class, this is to be signaled by another task on a given event. The EventDemo will simply loop around counting how many times it gets woken up. Data protection is demonstrated here by making both the smx_EventQueue object ec a protected member, as well as the counter member.

smx_EventQueue::smx_EventQueue(const char *name = NULL)

Constructs an event queue.

smx functions called smx_ EventQueueCreate()

Returns none

Example smx_EventQueue * ecp = new smx_EventQueue("MyEvent");

smx_EventQueue::~smx_EventQueue()

The destructor resumes all waiting tasks with FALSE return values then destroys this event queue. and deletes the underlying smx event queue.

smx functions called smx_EventQueueDelete()

Returns none

Errors none

Example

```
{
    smx_EventQueue  *ecp = new smx_EventQueue();
    // ...
    delete ecp;
}
```

BOOLEAN **smx_EventQueue::Count**(u32 count, u32 timeout = INF)

Suspends the current task on this event queue for *count* signals. *timeout* is the number of ticks the task will wait for the count to complete. The default timeout is infinite. If the count is not completed before the timeout occurs, the task is resumed with a FALSE return.

smx functions called smx_EventQueueCount()

Returns TRUE count completed.
FALSE error, 0 count or timeout, or timed out.

Example ecp->Count(10);

void **smx_EventQueue::CountStop**(u32 count, u32 timeout = INF)

Works like Count() except that the calling Task is first stopped. Then operation is similar to Count, except that the task restarts instead of resuming and par is the parameter passed to the task main function.

smx functions called smx_EventQueueCountStop()

par TRUE count completed.
FALSE error, 0 count or timeout, or timed out.

Example ecp->CountStop(10);

BOOLEAN **smx_EventQueue::Clear**(void)

Resumes all waiting tasks with FALSE returns, then clears the event queue. See the smx Reference Manual for an example.

smx functions called smx_EventQueueClear()

Returns TRUE queue cleared
FALSE error

smx_Msg Class

file xmsg.cpp

#include xmsg.hpp

derived from smx_Object class

smx_Msg objects (*messages*) are said to be either *holding* or *non-holding*. Holding means that it has an actual smx message; non-holding means that it does not. A non-holding message is necessary for receiving from an exchange. A holding message is necessary for sending to an exchange. After sending, the message becomes non-holding. Constructors are provided to create both types of messages.

Protected Data Members.

```
u32         ObjectFlags;
MCB_PTR     MsgCBP;
```

Constructors and Destructor

```
smx_Msg();
smx_Msg( u32 size );
smx_Msg( MCB_PTR  msg );
smx_Msg( smx_Pool &pool, u16 clrsz = 0 );
virtual ~smx_Msg();
```

Methods

```
BOOLEAN          Bump( u8 pri );
BOOLEAN          Holding();
BOOLEAN          Make( u32 size );
BOOLEAN          Make( smx_Pool *pool, u32 clrsz = 0 );
u32              Peek( SMX_PK_PARM par );
BOOLEAN          Rel( u16 clrsz = 0 );
BOOLEAN          Reply( smx_Msg *msg = NULL );
virtual SPP_CLTYPE WhatIs( void ); // returns SPP_CL_MSG
friend class     smx_MsgXchg;
```

smx_Msg::smx_Msg()

Constructs a non-holding message that is ready to receive an smx message from an exchange.

smx functions called none

Returns none

Example

```
TaskA::Main( u32 par )
{
      smx_Msg  MyMsg;
}
```

This example creates a message that is not holding. Its scope is within TaskA.

smx_Msg::smx_Msg(u32 bsize)

Create a holding message by getting a data block from the smx heap, making it into an smx message, and constructing this message. This message can then be filled with data and sent to an exchange.

smx functions called smx_HeapMalloc() and smx_MsgMake()

Returns none

Example

```
const u32  MSIZE = 5;
smx_MsgXchg  errorX; // global error exchange

void TaskA::Main( u32 par )
{
      void * dp;
      char * sp = "Err9";
      BOOLEAN  eflag;  // error flag

      // process data. set eflag if an error occurs.

      if( eflag )
      {
            smx_Msg  emsg( MSIZE );
            dp = (void *)emsg.Peek(SMX_PK_BP);
            memcpy( dp, sp, MSIZE );
            errorX->Send(&emsg);
      }
}
```

This example shows creating a holding error message, emsg, which exists only within the scope of the if statement. emsg is loaded with the error number and sent to errorX, the global error exchange. After the send, emsg is non-holding; it is automatically destroyed when it goes out of scope.

smx_Msg::smx_Msg(MCB_PTR msg)

Constructs a message from an existing smx message. This is useful when interfacing to smx code. It is protected so that it can only be used within a class derived from smx_Msg. or in a friend class or friend function. Since an existing message is used, no clrsz is specified.

smx functions called none

Returns none

Example

```
smx_Msg  *cp_msg;
MCB_PTR  c_msg;

App::Init()
{
      u8*   bp = (u8 *)smx_HeapMalloc(100);
      c_msg = smx_MsgMake(NULL, bp);
      ...
}

TaskA::Main( u32 par )
{
      cp_msg  smx_Msg( c_msg );
      while (1)
      {
            // use cp_msg
      }
}
```

In this example an smx message, c_msg, of 100 bytes is created, during initialization with a message block from the heap. It is made into an smx++ message, cp_msg, when TaskA starts and used within the TaskA main loop. cp_msg is deleted if TaskA is exited.

smx_Msg::~smx_Msg()

Destroys this message and deletes the underlying smx message, if holding and not converted from an smx message. If this message was converted from an smx message, then the smx message can be deleted only by smx.

smx functions called smx_MsgUnmake(), smx_HeapFree(), smx_MsgRel()

BOOLEAN **smx_Msg::Bump**(u8 pri)

This method is used to change a message's priority. If already in a queue, it is requeued after other messages of the new priority pri.

smx functions called smx_MsgBump()

Returns TRUE message priority changed
FALSE message priority not changed due to an error

Errors SMXE_NOT_HOLDING

Example
```
TaskA::Main( u32 par )
{
    MyMsgP->Bump( MyMsgP->Peek(SMX_PK_PRI)+1 );
}
```

In this example the priority of MyMsgP is incremented by 1.

BOOLEAN **smx_Msg::Holding**()

Reports if this message is holding.

smx functions called none

Returns TRUE holding
FALSE not holding

Example See example for smx_MsgXchg::Receive() method.

BOOLEAN **smx_Msg::Make**(u32 bsize)

Makes a non-holding message into a holding message by getting a data block from the heap, making it into an smx message, and linking the smx message to this message. This message can then be filled and sent to an exchange. This make is useful for messages of varying sizes.

smx functions called smx_HeapMalloc() and smx_MsgMake()

Returns TRUE this message is now holding
FALSE operation not competed due to an error

Errors SMXE_HOLDING

Example
```
smx_MsgXchg  xchgA; // global message exchange
```

```
TaskA::Main( u32 par )
{
      u8*  dp;
      smx_Msg  MyMsg; // create a non-holding msg

      if( MyMsg.Make( 50 ) )
      {
            dp = (u8*)MyMsg.Peek( SMX_PK_BP );
            // load data into Msg using dp
            xchgA.Send(MyMsg);
      }
      else
            // ... handle error
}
```

Initially, MyMsg is not holding a message. Make() gets a message from the heap for it to hold. Then, MyMsg is filled with data and sent to the global exchange, xchgA. At this point, MyMsg becomes non-holding, and the process repeats.

BOOLEAN **smx_Msg::Make**(smx_Pool *pool, u32 clrsz = 0)

Makes a non-holding message into a holding message by getting a data block from a block pool, making it into an smx message, and linking the smx message to this message. This message can then be filled and sent to an exchange. This make is useful for a messages of the size. Getting Messages from a pool is generally faster and more deterministic than from the heap.

smx functions called smx_MsgGet()

Returns

TRUE	this message is now holding
FALSE	operation not competed due to an error

Errors SMXE_HOLDING

Example

```
smx_MsgXchg  xchgA;      // global message exchange
smx_Pool  mypool(10, 50);       // global block pool

TaskA::Main( u32 par )
{
      u8*  dp;
      smx_Msg  MyMsg; // create a non-holding msg

      if( MyMsg.Make( mypool ) )
      {
            dp = (u8*)MyMsg.Peek( SMX_PK_BP );
            // load data into Msg using dp
            xchgA.Send(MyMsg);
      }
      else
            // ... handle error
}
```

Initially, MyMsg is not holding a message. Make() gets a message from mypool for it to hold. Then, MyMsg is filled with data and sent to the global exchange, xchgA. At this point, MyMsg becomes non-holding, and the process repeats.

u32 **smx_Msg::Peek(** SMX_PK_PARM par)

Returns the value of the specified parameter for this message. See smx_MsgPeek() in the smx Reference Manual for a list of message parameters.

smx functions called smx_MsgPeek()

Returns value Value of par.
0 No value, if error, or 0 value if not.

Errors SMXE_NOT_HOLDING

Example
```
smx_Msg  msg(200);
u8 *bp = (u8*)msg.Peek( SMX_PK_BP );
```

BOOLEAN **smx_Msg::Rel**(u32 clrsz = 0)

Makes this message into a non-holding message by releasing its smx message either to its block pool or to the heap depending on where it came from. This can be used for a message which was obtained through either holding constructor or through either make().

smx functions called smx_MsgPeek(), smx_MsgUnmake(), smx_HeapFree(), and smx_MsgRel()

Returns TRUE message successfully released
FALSE error

Errors SMXE_NOT_HOLDING

Example
```
MyTask::Main( u32 par )
{
    smx_Msg msg(100); // local message
    //…
    msg.Rel();
}
```

BOOLEAN **smx_Msg::Reply**()

If this message is non-holding, reports error and returns FALSE. If this message is holding and its reply field is an exchange, attempts to send this message to it. If not, returns FALSE. If send is successful, message becomes non-holding and returns TRUE. Otherwise, FALSE is returned.

smx functions called smx_MsgSend(), smx_SysWhatIs()

Returns
TRUE message successfully sent
FALSE message not sent due to error or not holding

Errors SMXE_NOT_HOLDING

Example

```
smx_MsgXchg *data_xchg = new smx_MsgXchg();

MyTask::Main( u32 par )
{
    u8 *        dp;
    smx_Msg *mp;

    while (mp = data_xchg->Receive())
    {
        dp = (u8*)msg->Peek(SMX_PK_BP);
        if (process_msg(dp))
            *dp = ACK;
        else
            *dp = NAK;
        mp->Reply();
    }
}
```

In this example, MyTask waits at data_xchg for incoming messages. When a message is received, it gets a pointer to its data block and calls process_msg(dp) to extract the data from the message and process it. It then reuses the message to send NAK or ACK to a reply exchange. Normally, the original sender would be waiting at this exchange to either resend the message or to send a new message.

smx_MsgXchg Class

file xmsgx.cpp

#include xmsgx.hpp

derived from smx_Object class

Exchanges provide a means whereby information may be transferred between tasks. The smx_MsgXchg class can be created as either a normal, pass, or broadcast type.

Protected Data Members

u32 **ObjectFlags**;
XCB_PTR **XchgCBP**;

Constructors and Destructor

smx_MsgXchg(SMX_XCHG_MODE mode = SMX_XCHG_NORM, const char *name = NULL);
virtual **~smx_ MsgXchg** ()

Methods

BOOLEAN **Clear**();
smx_Msg * **Receive**(u32 timeout = INF);
void **ReceiveStop**(u32 TimeOut);
BOOLEAN **Send**(smx_Msg &msg, u8 pri = 0, smx_MsgXchg *reply = NULL);
friend class **smx_Msg**;
virtual SPP_CLTYPE **WhatIs**(void); // returns SPP_CL_XCHG

smx_ MsgXchg::smx_ MsgXchg (SMX_XCHG_MODE mode = SMX_XCHG_NORM, const char *name = NULL)

Constructs a message exchange, which operates in the selected mode, as follows:

mode	exchange
SMX_XCHG_NORM	Normal
SMX_XCHG_PASS	Pass
SMX_XCHG_BCST	Broadcast

smx functions called smx_MsgXchgCreate()

smx_ MsgXchg::~smx_ MsgXchg ()

Resumes all waiting tasks with 0 return values or releases all waiting messages. Then deletes the underlying smx message exchange and destroys this message exchange.

smx functions called smx_MsgXchgDelete()

Errors SMX_OBJ_NOT_CREATED

BOOLEAN **smx_MsgXchg::Clear**()

Resumes all waiting tasks with 0 return values or releases all waiting messages. Then clears appropriate fields in the underlying smx message exchange.

smx functions called smx_MsgXchgClear()

Returns TRUE exchange cleared or already clear
FALSE error

Errors SMX_OBJ_NOT_CREATED

smx_Msg ***smx_MsgXchg::Receive**(u32 timeout = INF)

Returns a holding message from this exchange. If the exchange has no messages waiting, suspends the current task for timeout ticks. Will return to the current task with 0 if timeout elapses before a message arrives or if there is an error.

In order to get a message, first receives an smx message from the underlying smx exchange, then creates a new message using the smx_Msg conversion constructor. To do this, smx_MsgXchg must be a friend class to the smx_Msg class.

smx functions called smx_MsgReceive()

Returns smx_Msg* message pointer
NULL timeout or error

Errors SMX_OBJ_NOT_CREATED

Example

```
class MyTask : public smx_Task
{
    smx_MsgXchg  in_xchg;
    smx_Msg*  msgp;
```

```
        void Main( u32 par )
        {
            while (msgp = in_xchg.Receive(INF))
            {
                /* process msg */
                delete msgp;
            }
        }
}
```

This example defines a new task class which receives messages, processes them, then deletes them.

smx_Msg ***smx_MsgXchg::ReceiveStop**(u32 timeout = INF)

This method works like Receive(), but it first stops the current task. As soon as a message is received from this exchange, the task is restarted.

smx functions called smx_ReceiveStop()

Returns none

par handle Message handle received.
NULL Error or timeout.

Errors SMX_OBJ_NOT_CREATED

Example

```
class MyTask : public smx_Task
{
    smx_MsgXchg  in_xchg;

    void Main( u32 par )
    {
        if (par > 0)
        {
            smx_Msg *msgp = new smx_Msg((MCB_PTR)par);
            // process msg */
            delete msgp;
        }
        in_xchg.ReceiveStop(INF)
    }
}
```

This example does the same thing as the previous example. MyTask is started the first time with par = 0, so it goes immediately to receive stop from in_xchg. When it receives a message, it starts again and constructs an smx++ message from the message handle passed in to its main function. It then processes the message, deletes it, and wait stops for the next message.

BOOLEAN **smx_MsgXchg::Send**(smx_Msg *mp, u8 pri = 0, smx_MsgXchg *reply = NULL)

Sends a holding message to this exchange with priority pri. On successful send returns TRUE. See the smx Reference Manual section on smx_MsgSendPR() for more details.

smx functions called smx_MsgSendPR()

Returns

TRUE	message sent
FALSE	error

Errors SMX_OBJ_NOT_CREATED, SMXE_NULL_PTR_REF, SMXE_NOT_HOLDING

Example

```
typedef struct
{
      u32 hdr;
      u8   data[16];
} *MB_PTR;

smx_Pool free_msgs = smx_Pool(10, 20);
smx_Msg msg = smx_Msg();
smx_MsgXchg xchg = smx_MsgXchg();

BOOLEAN SendMsg(void)
{
      MB_PTR    dp;

      mp->Make(free_msgs);
      dp = (MB_PTR)mp.Peek( SMX_PK_BP );
      dp->hdr = TEST;
      for (i = 0; i < 16; i++)
            dp->data[i] = i;
      return xchg.Send(mp);
}
```

In this example, free _msgs is a pool of 10 message blocks of 20 bytes, each. msg is a non-holding message and xchg is a normal message exchange. SendMsg() makes msg into a holding message from free_msgs, fills it with a test pattern, and sends it to xchg. The priority and reply of msg are 0. SendMsg() returns TRUE, if a message is sent, FALSE otherwise.

smx_Mutex Class

file xmtx.cpp

#include xmtx.hpp

derived from smx_Object class

The smx_Mutex class is used to create a mutex, possibly with priority inheritance enabled, and/or ceiling priority.

Protected Data Members

u32 **ObjectFlags**;
MUCB_PTR **MutexCBP**;

Constructors and Destructor

smx_Mutex(u8 pi, u8 ceiling, const char *name = NULL);
virtual ~**smx_Mutex**();

Methods

BOOLEAN **Clear**();
BOOLEAN **Free**();
BOOLEAN **Get**(u32 timeout = INF);
void **GetStop**(u32 timeout = INF);
BOOLEAN **Rel**();
virtual SPP_CLTYPE **WhatIs**(void); // returns SPP_CL_MTX

Example

```
class ConIO : public smx_Mutex
{
public:
      ConIO() : smx_Mutex(1, 0) { };

      void write(char *str)
      {
            if (Get())
            {
                  sb_ConWriteStringUnp(0, DEMO_ROW + 2, SB_CLR_LIGHTBLUE,
                        SB_CLR_BLACK, !SB_CON_BLINK, str);
                  Rel();
             }
      }
}  IOM;
```

This example demonstrates how to make a shared resource object which is controlled by an smx_Mutex object. Here we are ensuring that only one task will write to the console at a time. This object is derived from smx_Mutex so it inherits all of the features of a Mutex object. We have added a public write() method which gets access through the mutex, writes to the console, and then releases it. The object instance, IOM, will be constructed and destroyed by compiler startup and exit code since it is a globally defined object. The ConIO objects will all have priority inheritance enabled and ceiling priority disabled. The IOM object is used as follows:

```
IOM.write("Hello world!");
```

smx_Mutex::smx_Mutex(u8 pi, u8 ceiling, const char *name = NULL)

Constructs a mutex with priority inheritance if pi > 0 and with ceiling priority if ceiling > 0.

smx functions called smx_MutexCreate()

Example

```
smx_Mutex * MutexA = new smx_Mutex(1, 0, "MutexA" );
```

MutexA is created with priority inheritance enabled and ceiling disabled.

smx_Mutex::~smx_Mutex()

Resumes all waiting tasks with FALSE, deletes the underlying smx mutex, and destroys this mutex. See smx_MutexDelete in the smx Reference manual for more details.

smx functions called smx_MutexDelete().

Example

```
smx_Mutex * MutexA = new smx_Mutex(1, 0, "MutexA" );
…
delete MutexA;
```

BOOLEAN **smx_Mutex::Clear**()

Removes this mutex from the current owner's mutex queue and adjusts the priority of the current owner to its highest owned-mutex priority or to its normal priority, if none. Clears the task queue of this mutex by resuming all tasks in it with FALSE returns. Then puts this mutex in its cleared state. Operations are performed regardless of mutex owner and nesting count. This method is normally used for recovery purposes.

smx functions called smx_MutexClear()

Returns TRUE mutex cleared
FALSE error

Errors SMX_OBJ_NOT_CREATED

Example

```
smx_Mutex MutexA = new smx_Mutex( 1, 0, “MutexA” );

void SysRecover(void)
{
      …
      MutexA->Clear();
      ...
}
```

In this example, one of the things that the system recovery routine does is to clear the global MutexA, in order to put the system back into its ground state.

BOOLEAN **smx_Mutex::Free**()

Removes this mutex from the current owner task’s mutex queue, adjusts the priority of the task to its highest owned-mutex priority or to its normal priority, if none, and resumes the task with FALSE. Then makes the next waiting task the new owner or puts this mutex into the free state, if no task is waiting. Differs from Rel() in that the task calling this method need not be the owner of this mutex. Differs from Clear() in that it does not clear the task wait queue.

smx functions called smx_MutexFree()

Returns TRUE mutex freed
FALSE error

Errors SMX_OBJ_NOT_CREATED

Example

```
smx_Mutex *MutexA = new smx_Mutex( 1, 0, “MutexA” );

TaskA::Main( u32 par )
{
      MutexA->Get()
      …
      MutexA->Free();
}
```

BOOLEAN **smx_Mutex::Get**(u32 timeout = INF)

Gets this mutex, if free, and returns TRUE. Otherwise waits until this mutex becomes available or timeout occurs. If current task already owns this mutex, increments its nesting counter and returns TRUE.

smx functions called smx_MutexGet()

Returns TRUE mutex obtained or already owned.

FALSE error or timeout.

Errors SMX_OBJ_NOT_CREATED

Example See smx_Mutex example.

void **smx_Mutex::GetStop**(u32 timeout = INF)

Same as Get() method, except that the current task is stopped and then restarted instead of resumed.

smx functions called smx_MutexGetStop()

Returns none

par TRUE mutex obtained or already owned.
FALSE error or timeout.

Errors SMX_OBJ_NOT_CREATED

Example

```
smx_Mutex MtxA = new smx_Mutex(1, 0);

void TaskA_Main(par ok)
{
    if (ok)
    {
        // access protected resource
        MtxA->Rel();
    }
    MtxA->GetStop(10);
}
```

The first time TaskA is started, par == 0, so it will try to get MtxA. If it is not free, TaskA will wait for it, with no stack. When TaskA gets mtxA, it is restarted and the result of GetStop() is passed as par to its main function. If par == TRUE, TaskA will access the resource, then release MtxA. Then it will try again.

BOOLEAN **smx_Mutex::Rel**()

Releases this mutex, if owned by the current task and the nesting count == 1. If owned, but the nesting count is > 1, decrements it and returns TRUE. If not owned, fails and returns FALSE.
In the first case, removes this mutex from the current task's mutex list, adjusts the priority of the task to its highest owned-mutex priority or to its normal priority, if none, and resumes the task with TRUE. Then the next waiting task becomes the new owner.

smx functions called smx_MutexRel()

Returns

TRUE	mutex released
FALSE	error

Errors SMX_OBJ_NOT_CREATED

Example See example above.

smx_Pipe Class

file xpipe.cpp

#include xpipe.hpp

derived from smx_Object class

The smx_Pipe class is used in the same manner as an smx pipe. Pipes are used for I/O or for communication between tasks. Data packets can be any width from 1 to 255 bytes. Pipes are good for low-speed data transfers.

Protected Data Members

u32 **ObjectFlags**;
PICB_PTR **PipeCBP**;

Constructors and Destructor

smx_Pipe(u8 width, u16 length, const char *name = NULL);
virtual ~**smx_Pipe**();

Methods

BOOLEAN **Clear**();
BOOLEAN **Get**(void *pdst);
BOOLEAN **Get8**(u8 *bp);
BOOLEAN **GetWait**(void *pdst, u32 timeout = INF);
void **GetWaitStop**(void *pdst, u32 timeout = INF);

BOOLEAN **Put**(void *psrc);
BOOLEAN **Put8**(u8 byte);
BOOLEAN **PutWait**(void *psrc, u32 timeout = INF);
void **PutWaitStop**(void *psrc, u32 timeout = INF);

BOOLEAN **Resume**();
u32 **Status**(PSS *ppss);
virtual SPP_CLTYPE **WhatIs**(void); // returns SPP_CL_PIPE

smx_Pipe::smx_Pipe(u8 width, u16 length, const char *name = NULL)

Constructs a pipe of the specified width and length from the heap.

smx functions called smx_HeapMalloc() and smx_PipeCreate()

Returns none

Example

```
smx_Pipe *MyPipeP;

void App_Init(void)
{
    ...
    MyPipeP = new smx_Pipe( 4, 25, “MyPipe” );
    ...
}
```

In this example, the application initialization function creates MyPipe with 25 cells of 4-byte width. The new operator uses the smx_Pipe constructor and returns a pointer to MyPipe. The pipe buffer is obtained from the heap and the pipe object is obtained from the object pool.

smx_Pipe::~smx_Pipe()

Resumes all waiting tasks with FALSE return values, deletes the underlying smx pipe and frees its packet space back to the heap, then destroys this pipe.

smx functions called smx_PipeDelete() and smx_HeapFree()

Returns none

Example

```
void App_Exit(void)
{
    ...
    delete MyPipeP;
    ...
}
```

In this example, the application exit function deletes MyPipe. Since this pipe was created with the new operator, it must be explicitly deleted. delete calls the destructor for MyPipe. The pipe buffer is returned to the heap and the pipe object is returned to the object pool.

BOOLEAN **smx_Pipe::Clear**()

Resumes all tasks waiting on this pipe with FALSE, then put the pipe in its initial state.

smx functions called smx_PipeClear()

Returns

TRUE	Pipe cleared.
FALSE	Error.

Errors SMX_OBJ_NOT_CREATED

Example

```
void App_Recovery(void)
{
    ...
    MyPipeP->Clear();
    ...
}
```

In this example, the application recovery function clears MyPipe in order to start over.

BOOLEAN **smx_Pipe::Get**(void *pdst)

Gets the next packet from this pipe and loads it into the buffer at pdst. Does not wait if the pipe is empty, just returns immediately with FALSE. Intended for I/O transfers between tasks and ISRs or LSRs. Does not wake up a task waiting to put a packet into this pipe. See the smx Reference Manual for operational details.

smx functions called smx_PipeGet() — bare function.

Returns

TRUE	Packet transferred.
FALSE	Packet not transferred due to error.

Errors SMX_OBJ_NOT_CREATED

Example

```
class TaskA : public smx_Task
{
    ...
    virtual void Main( u32 par )
    {
        u32 mb[16];
        u32 i;

        for (i = 0; i < 16; i++)
        {
            MyPipeP->Get(mb++))
        {
        // process mb
    }
};
```

The Main method of TaskA gets 16 words from MyPipe and puts them into mb. It then processes the data in mb and stops. MyPipe is a global object filled by another task or LSR. When the pipe has at least 16 words in it, TaskA is restarted (not shown) to get and process them.

BOOLEAN **smx_Pipe::Get8**(u8 *bp)

Gets the next byte from this pipe and loads it at bp. Does not wait if the pipe is empty, just returns immediately with FALSE. Intended for I/O transfers between tasks and ISRs or LSRs. Does not wake up a task waiting to put a byte into this pipe. See the smx Reference Manual for operational details. This is the fast version of Get() for byte use.

smx functions called smx_PipeGet8() — bare function

Returns TRUE byte transferred.
FALSE byte not transferred.

Errors SMX_OBJ_NOT_CREATED

Example

```
smx_Pipe *MyPipeP = new smx_Pipe(1, 64);

class TaskA : public smx_Task
{

    virtual void Main( u32 par)
    {
        u8 mb[16];
        u32 i;

        for (i = 0; i < 16; i++)
        {
            MyPipeP->Get(mb++))
        {
        // process mb
    }
};
```

This is similar to the previous example, except that a byte-wide pipe is used and only one byte, at a time, is transferred to mb.

BOOLEAN **smx_Pipe::GetWait**(void *pdst, u32 timeout = INF)

If this pipe is not empty, transfers the oldest packet from it to the buffer at pdst and returns TRUE. If this pipe is empty, waits up to timeout ticks for a packet, then returns FALSE, if none arrive. Intended for task to task communication. Will wake up a task waiting to put a packet into a full pipe. See the smx Reference Manual for operational details. Can be used from LSRs, but they cannot wait on pipes.

smx functions called smx_PipeGetWait()

Returns TRUE Packet transferred.
FALSE Packet not transferred.

Errors SMX_OBJ_NOT_CREATED

Example

```
smx_Pipe *in_pipeP = new smx_Pipe(PSIZE, 10);

class TaskA : public smx_Task
{
    ...
    virtual void Main( u32 par )
    {
        u8 pkt[PSIZE];

        while (in_pipeP->GetWait(pkt))
        {
            // process pkt
        }
    }
};
```

The Main method of TaskA waits on in_pipe for a packet. When a packet is received, it is put into pkt and processed. Then TaskA waits for the next packet. It will wait indefinitely. This creates synchronization between TaskA and whatever task is putting packets into in_pipe. Note that in_pipe is global, so both tasks can access it.

void **smx_Pipe::GetWaitStop**(void *pdst, u32 timeout = INF)

Same as GetWait() method, except that the current task is stopped and then restarted instead of resumed. Can be used from a task, but not from an LSR.

smx functions called smx_PipeGetWaitStop()

par

TRUE	Packet transferred.
FALSE	Packet not transferred.

Errors SMX_OBJ_NOT_CREATED

Example

```
class KeyTask : public smx_Task
{
public:

    KeyTask() : smx_Task(2)
    {
        is_starting = TRUE;
        key_pipe = new smx_Pipe(1, 40);  /* byte wide pipe */
    }
```

```
        void Put8(u8 ch)
        {
                key_pipe->Put8(ch);
        }

        virtual void Main( u32 got_key )
        {
                if (!is_starting)
                {
                        if (got_key)
                                process_key(key_buf);
                        else
                                report_problem();
                }
                else
                        is_starting = FALSE;  /* wait for first key */
                key_pipe->GetWaitStop((void *)key_buf, 100);
        }

private:
        BOOLEAN         is_starting;
        u8              key_buf[40];
        smx_Pipe        *key_pipe;

        void process_key(u8 *bp)   {  /* … */ }
        void report_problem()     {  /* .. */  }
} key_task;

void key_LSR(u32 par)
{
        key_task.Put8((u8)par);
}
```

The Main method of KeyTask waits for a key on its key_pipe. Note that the is_starting flag is used to skip processing on the initial startup. Since we are using GetWaitStop(), TRUE will be passed as the got_key parameter to Main() if a key was successfully placed into the key buffer (key_buf). got_key will be false if 100 ticks have passed without a key or if there was an error, in which case report_problem() is called.

This demonstrates the value of smx++'s support for dual API programming. In this case the LSR (and ISR) are written in C code, while the key processing task is written with smx++. Note how the KeyTask offers a public Put() method which anybody can use to put a character into the key_pipe. The key_LSR can be invoked from an ISR where the key is passed as a parameter.

WARNING: It is never a good idea to call any scheduling method in the constructor of a task such as Stop, Start, Suspend, etc. This will most likely cause an error since the task is still "under construction".

BOOLEAN **smx_Pipe::Put**(void *psrc)

Puts the packet from the buffer at psrc into pipe. Does not wait if the pipe is full, just returns immediately with FALSE. Intended for I/O transfers between ISRs or LSRs and tasks. Does not wake up a task waiting to get a packet from this pipe. See the smx Reference Manual for operational details.

smx functions called smx_PipePut() — bare function.

Returns TRUE Packet transferred.
FALSE Packet not transferred.

Errors SMX_OBJ_NOT_CREATED

Example

```
smx_Pipe * MyPipeP = new smx_Pipe(4, 20);
u32 packet;

TaskA::Main( u32 par )
{
    // …
    MyPipeP->Put(&packet);  /* does not wait */
    // …
}
```

BOOLEAN **smx_Pipe::Put8**(u8 byte)

Puts byte into this pipe. Does not wait if the pipe is full, just returns immediately with FALSE. Intended for I/O transfers between ISRs or LSRs and tasks. Does not wake up a task waiting to get a byte from this pipe. See the smx Reference Manual for operational details. This is the fast version of Put() for byte use.

smx functions called smx_PipePut() — bare function

Returns TRUE byte put into pipe.
FALSE byte not put into pipe.

Errors SMX_OBJ_NOT_CREATED

Example See example for GetWaitStop()

BOOLEAN **smx_Pipe::PutWait**(void *psrc, u32 timeout = INF)

If this pipe is not full, transfers the packet at psrc to it and returns TRUE. If this pipe is full, waits up to timeout ticks for an empty slot, then returns FALSE, if none appears. Intended for task to task communication. Will wake up a task waiting to get a packet from an empty pipe. See the smx Reference Manual for operational details. Can be used from LSRs, but they cannot wait on pipes.

smx functions called smx_PipePutWait()

Returns TRUE Packet transferred.
FALSE Packet not transferred.

Errors SMX_OBJ_NOT_CREATED

Example

```
TaskA::Main( u32 par )
{
    ...
    while (MyPipeP->PutWait(&packet))
    {
        // prepare another packet.
    }
}
```

Puts a 4-byte packet into MyPipe. Waits indefinitely if this pipe is full.

void **smx_Pipe::PutWaitStop**(void *psrc, u32 timeout = INF)

Same as PutWait() method, except that the current task is stopped and then restarted instead of resumed. Can be used from a task, but not from an LSR.

smx functions called smx_PipePutWaitStop()

par TRUE Packet transferred.
FALSE Packet not transferred.

Errors SMX_OBJ_NOT_CREATED

Example

```
class CrtTask : public smx_Task
{
public:
    CrtTask() : smx_Task(2);  { crt_pipe = new smx_Pipe(4, 10); }

    void Main(u32 par)
    {
        // create packet @ pktp
        crt_pipep->PutWaitStop(pktp);
    }
```

```
    BOOLEAN GetPkt { crt_pipep->GetWait(); }

private:
    smx_Pipe    *crt_pipep;
    u8          pktp[4];
} crt_task;
```

crt_task is constructed with a priority of 2 and its private member pipe, crt_pipe, is constructed with its data buffer from the heap. The crt_task main function creates a packet, loads it into crt_pipe, then stops and restarts. It will wait to restart if the pipe is full. A public method, GetPk() is provided to get packets out of crt_pipe. It will wait if crt_pipe is empty.

BOOLEAN **smx_Pipe::Resume**()

Resumes each task waiting on this pipe, for which its put or get condition is TRUE. This method is used in combination with non-SSR methods such as Get8() and Put8().

smx functions called smx_PipeResume()

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

Errors SMX_OBJ_NOT_CREATED

Example

```
smx_Pipe MyPipe(1, 80); // byte wide, 80 character length

void key_LSR(void)
{
    MyPipe.Resume();
}

void key_ISR(u8 key_port)
{
    u8 ch;
    ch = input_key(key_port);
    MyPipe.Put8(ch);
    smx_LSR_INVOKE(key_LSR, 0);
}
```

In this example, key_ISR is loading MyPipe, 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 MyPipe.Resume(). Note that key_LSR does not need to know what task is waiting, if any.

u32 **smx_Pipe::Status**(PSS *ppss)

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

smx functions called smx_PipeStatus()

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

Errors SMX_OBJ_NOT_CREATED

Example

```
smx_PipeP MyPipe;

TaskA::Main(u32 par)
{
    PSS MP_Info;
    u32  MP_Num;

    MP_Num = MyPipe->Status(&MP_Info);
    ...
}
```

In this example the number of packets in the pipe is put into MP_Num and other pipe status information is put into MP_Info.

smx_Pool Class

file xpool.cpp

#include xpool.hpp

derived from smx_Object class

The smx_Pool class provides a C++ interface to an smx block pool, which can be obtained from the heap or a DAR. smx block pools are discussed in more detail in the smx Reference Manual.

Protected Data Members

u32 **ObjectFlags**;
PCB_PTR **PoolCBP**;

Constructors and Destructor

smx_Pool(u32 numblks, u32 blksize, const char * name = NULL);
smx_Pool(SB_DCB_PTR dar, u32 numblks, u32 blksize,
u16 align = SB_CACHE_LINE, const char * name = NULL);
virtual ~**smx_Pool**();

Methods

u32 **Peek**(SMX_PK_PARM par);
virtual SPP_CLTYPE **WhatIs**(void); // returns SPP_CL_POOL

Example

```
class WorkAreas : smx_Pool, smx_BLock;
{
public:
    WorkAreas()  smx_Pool(10, 40, "workareas") {};
    ~WorkAreas() ~smx_Pool();

    BCB_PTR Get() { return( new smx_Block( ) )
} wa;

BCB_PTR wa.Get();
```

In this example, a WorkAreas class is defined which includes a protected pool that is created by the constructor and can be deleted by the destructor. A single method, Get() is provided to get a work area using the smx_Block constructor.

smx_Pool::smx_Pool(u32 NumBlks, u32 BlkSize, const char *name = NULL)

Constructs a block pool of NumBlks of BlkSize bytes from the heap.

smx functions called smx_HeapMalloc() and smx_BlockPoolCreate()

Returns none

Example

```
smx_Pool * MyPoolP  = new smx_Pool( 10, 100, “MyPool” );
```

Creates MyPool consisting of 10 blocks of 100 bytes each from the heap.

smx_Pool::smx_Pool(SB_DCB_PTR dar, u32 NumBlks, u32 BlkSize, u16 align, const char *name = NULL)

Creates a block pool of NumBlks of BlkSize from dar aligned on an align-byte boundary.

smx functions called smx_BlockPoolCreateDAR()

Returns none

Example

```
smx_Pool *MyPoolP  = new smx_Pool( sb_adar, 10, 100, 16, “MyPool” );
```

Creates MyPool consisting of 10 blocks of 100 bytes, each, from ADAR with 16-byte alignment.

smx_Pool::~smx_Pool()

Releases all the memory used by this pool back to the heap or to the dar from which it came, then destroys this pool, unless one or more blocks are in use.

smx functions called smx_BlockPoolDelete()

Returns none

Errors SMXE_OBJ_IN_USE

Example

```
smx_Pool * MyPoolP  = new smx_Pool( 10, 100, “MyPool” );
// use MyPool
delete MyPool;
```

u32 **smx_Pool::Peek**(SMX_PK_PARM par)

Returns the value of the specified parameter for this pool. See smx_BlockPoolPeek() in the smx Reference Manual for a list of pool parameters.

smx functions called smx_BlockPoolPeek()

Returns

value	Value for par.
0	Value, unless error.

Errors SMX_OBJ_NOT_CREATED

Example

```
u32 num_blocks = MyPoolP->Peek(SMX_PK_NUM);
```

This example shows how to find out how many blocks are in MyPool.

smx_Sem Class

file xsem.cpp

#include xsem.hpp

derived from smx_Object class

The smx_Sem class is used to create a semaphore with a specified threshold and one or more task priority levels. Semaphores are used to regulate the flow of tasks. smx_Sem is capable of operating in one of 6 modes: binary resource, multiple resource (counting semaphore), binary event, multiple event, threshold, and gate. For more information about these semaphore modes please refer to the Semaphores Chapter of the smx Users Guide.

Protected Data Members

u32 **ObjectFlags**;
SCB_PTR **SemCBP**;

Constructors and Destructor

smx_Sem(SMX_SEM_MODE mode, u32 lim = 1L, const char *name = NULL);
virtual **~smx_Sem()**;

Methods

BOOLEAN **Clear**();
u32 **Peek**(SMX_PK_PARM par);
BOOLEAN **Signal**();
BOOLEAN **Test**(u32 timeout = INF);
void **TestStop**(u32 timeout = INF);
virtual SPP_CLTYPE **WhatIs**(void); // returns SPP_CL_SEM

smx_Sem::smx_Sem(SMX_SEM_MODE mode, u32 lim = 1, const char *name = NULL)

Constructs a semaphore with mode and limit as specified:

mode	lim	Semaphore Mode
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

smx functions called smx_SemCreate()

Example

```
smx_Sem *PrintP = new smx_Sem( SMX_SEM_RSRC, 1, “Print” );
```

Constructs a binary resource semaphore, Print, which is used to limit one task, at a time, access to a printer.

smx_Sem::~smx_Sem()

Resumes all tasks waiting at this semaphore, with FALSE return values, then deletes the underlying smx semaphore, and destroys this semaphore.

smx functions called smx_SemDelete()

Example

```
PrintP->delete()
```

Destroys the Print semaphore above.

BOOLEAN **smx_Sem::Clear**()

Resumes all tasks waiting at sem with FALSE return values. Then resets the internal count to its initial value (lim). Useful in recovery situations.

smx functions called smx_SemClear()

Returns TRUE semaphore cleared
FALSE error

Errors SMX_OBJ_NOT_CREATED

Example

```
PrintP->Clear();
```

Clears the Print semaphore above.

u32 **smx_Sem::Peek**(SMX_PK_PARM par)

Returns the value of the specified parameter for this semaphore. See smx_SemPeek() in the smx Reference Manual for a list of semaphore parameters.

smx functions called smx_SemPeek()

Returns value value for par.
0 value unless error.

Errors SMX_OBJ_NOT_CREATED

Example

```
SMX_SEM_MODE mode = (SMX_SEM_MODE)PrintP->Peek(SMX_PK_MODE);
```

Returns the mode of the Print semaphore.

BOOLEAN **smx_Sem::Signal**()

Signals this semaphore that an event has occurred. Operation will depend on this semaphore mode and its internal count. For more discussion of how the various semaphore modes operate, refer to the smx Users Guide.

smx functions called smx_SemSignal()

Returns TRUE signal sent
FALSE error

Errors SMX_OBJ_NOT_CREATED

Example

```
smx_Sem *BinSemP = new smx_Sem(SMX_SEM_RSRC, 1, "bin_sem");

TaskA::Main( u32 par )
{
    while (1)
    {
        if (PrintP->Test(60))
        {
            // print report
            PrintP->Signal();
        }
        else
            // report printer busy
    }
}
```

In this example TaskA waits for an external signal before using the printer. If the signal does not arrive within 60 ticks, TaskA reports that the printer is busy, then waits again. When Print does pass, TaskA prints its report, then signals Print, so that another task can use the printer.

BOOLEAN **smx_Sem::Test**(u32 timeout = INF)

Test if this semaphore has a pass condition. If so, decreases its internal counter and continues execution of the current task. If the timeout elapses before a pass condition occurs, a waiting task is resumed with a FALSE. See smx_SemTest() in the smx Reference Manual for discussion of how the semaphore modes effect the pass condition.

smx functions called smx_SemTest()

Returns TRUE pass
FALSE error or timeout

Errors SMX_OBJ_NOT_CREATED

Example See smx_Sem::Signal() example.

void **smx_Sem::TestStop**(u32 timeout = INF)

Same as Test() method, except that the current task is stopped and then restarted instead of resumed. Can be used from a task, but not from an LSR.

smx functions called smx_SemTestStop()

Returns none

par TRUE pass
FALSE error or timeout

Errors SMX_OBJ_NOT_CREATED

Example

```
TaskA::Main( u32 ready )
{
    if( ready )
    {
        // print report
        PrintP->Signal();
    }
    else
    {
        // report printer busy
    }
    PrintP->TestStop( 60 );
}
```

This example does the same thing as the example for Signal(), except that TaskA waits in the stopped condition. This allows giving up its stack so it can be shared by other tasks.

smx_Task Class

file xtask.cpp

#include xtask.hpp

derived from smx_Object class

The task class is different from other smx++ classes, because application task classes derived from it will do most of the work. Normally, each derived task class will have its own main() function. The task Main() method is a pure virtual method since it must always be overridden in a derived task class. It basically determines what the task does. The HookEntry() and HookExit() methods are virtual methods, which must also be overridden, if they are used.

When dispatching a task, if the *this* pointer in its TCB is non-zero, the smx scheduler will call the task main wrapper, with the this pointer and the TCB return value, rv, as its parameters:

```
void smx_TaskMainWrapper(smx_Task *Task, u32 par)
```

This results in calling Main(par) for the task. The parameter, par, is the return value from Stop() methods described elsewhere in this manual. The parameter does not have to be used if not needed.

When suspending a task, if the this pointer in its TCB is non-zero, the smx scheduler will call the hook exit wrapper, with the this pointer as its parameter:

```
void smx_TaskHookExitWrapper(smx_Task *Task)
```

This results in automatically calling HookExit() for the task when it is suspended.

When resuming a task, if the this pointer in its TCB is non-zero, the smx scheduler will call the hook entry wrapper, with the this pointer as its parameter:

```
void smx_TaskHookEntryWrapper(smx_Task *Task)
```

This results in automatically calling HookEntry() for the task when it is resumed.

The above wrappers are defined in ptask.cpp.

Protected Data Members

u32 **ObjectFlags**;
TCB_PTR **TaskCBP**;

Constructors and Destructor

smx_Task(u8 pri, u32 stksz = 0, u32 flags = SMX_FL_NONE, const char *name = NULL);
virtual **~smx_Task**();

Methods

BOOLEAN **Bump**(u8 pri);
BOOLEAN **Hook**();
virtual void **HookEntry**();
virtual void **HookExit**();
void * **Locate**();

```
virtual void    Main(u32 par);
u32             Peek( SMX_PK_PARM par );
u32             RelAllBlocks();
u32             RelAllMsgs();
BOOLEAN         Resume();
BOOLEAN         StackCheckOff();
BOOLEAN         StackCheckOn();
BOOLEAN         Start();
BOOLEAN         Start( u32 par );
BOOLEAN         Stop( u32 timeout = INF );
BOOLEAN         Suspend( u32 timeout = INF );
BOOLEAN         UnHook();
virtual SPP_CLTYPE WhatIs( void ); // returns SPP_CL_TASK
```

Protected Methods

```
BOOLEAN         Lock();
BOOLEAN         LockClear();
BOOLEAN         Sleep( u32 stime );
void            SleepStop( u32 stime );
BOOLEAN         UnLock();
BOOLEAN         UnLockQuick();
```

These methods can be called only for the current task and hence they are protected.

Example

```
class Sleeper : public smx_Task
{
public:
    Sleeper( u8 pri ) : smx_Task( pri ), WakeUpTimer( NULL ) { };
    BOOLEAN  StopTimer( u32 timeout = INF );
    static void WakeUpLSR( u32 );

protected:
    virtual void Main( u32 par );      // Override inherited main method
    PTimer      WakeUpTimer;       // Timer used to activate task at regular intervals.
};

Sleeper   SleeperT((u8)PRI_NORM);   /* Sleeper task object */

// This LSR is associated with the smx_Timer object.  It will be activated at regular
//  intervals to restart the Sleeper task with a parameter of TRUE.

void Sleeper::WakeUpLSR( u32 )
{
    SleeperT.Start( (u32)TRUE ); // Restart task at regular intervals.
}
```

```
// This task demonstrates how to make a task periodically wake up.
// Uses WakeUpTimer and WakeUpLSR to wake up every 5 seconds.
// Set parameter DelayOk to TRUE unless you want to start the timer.

void Sleeper::Main( u32 DelayOk )
{
      UnLock();
      if (DelayOk)
            sb_ConWriteString(24, 9, LIGHTCYAN,BLACK,!SB_CON_BLINK,
                                          "WAKEUP");
      else
      {
            sb_ConWriteString(24,9,LIGHTCYAN,BLACK,!SB_CON_BLINK, "INIT  ");
            if (WakeUpTimer == NULL)
                  WakeUpTimer = new smx_Timer( (LSR_PTR)&Sleeper::WakeUpLSR,
                                          0, 0,  5*smx_cf.sec, "WakeUpTimer");
      }
}

// Enhance smx_Task::Stop to stop the timer as well.

BOOLEAN  Sleeper::StopTimer( u32 timeout )
{
      if (WakeUpTimer)
      {
            delete WakeUpTimer;
            WakeUpTimer = NULL;
      }
      return smx_Task::Stop( timeout );
}

App::Init()
{
      SleeperT.Start( (u32) FALSE ); // Pass parameter of FALSE.
}

App::ShutDown()
{
      SleeperT.Stop();
}
```

The above example starts the task SleeperT at regular intervals, each 5 seconds. The static method, WakeUpLSR, is the LSR invoked by the timer, WakeUpTimer. The LSR must be static because the smx scheduler does not setup the this pointer for LSRs. The parameter to the LSR is not utilized by this example. The Timer is started by the Sleeper::Main method which is passed a FALSE parameter and stopped by the virtual method Sleeper::Stop.

smx_Task::smx_Task(u8 pri, u32 stksz = 0, u32 flags = SMX_FL_NONE,
const char *name = NULL)

Constructs a task with the priority pri and specified flags set. If stksz is non-zero, a stack is obtained from the heap and bound to this task. Otherwise that task will receive a stack from the stack pool when it is dispatched.

smx functions called smx_TaskCreate()

Example

```
class PreemptorC : public smx_Task
{
    PreemptorC(u8 pri, u32 stksz, u32 flags, cons char *name) : smx_Task(pri, stksz,
                flags, name) {};
    void Main( u32 par );
}

PreemptorC::Main( u32 par )
{
    /* PreemptorC main code goes here */
}

PreemptorP = new PreemptorC( PRI_HI, 0, SMX_FL_NONE, "MyTask"  );
```

In this example, the PreemptorC class is derived from the smx_Task class and creates its own Main method, PreemptorC::Main(). The Preemptor constructor loads tcb.fun with the address of the C wrapper function that will call the PreemptorC main method and also loads tcb.thisptr with PreemptorP.

This is the typical procedure to create new tasks: derive a new task class from smx_Task, define its main method, then construct a task object. Task-specific hook entry and exit routines may also be defined. In some cases, multiple tasks may share a single derived task class.

BOOLEAN **smx_Task::Bump**(u8 pri)

Changes this task's priority and requeues it, if it is in a queue.

smx functions called smx_TaskBump()

Returns TRUE task priority changed
FALSE error

Errors SMX_OBJ_NOT_CREATED

Example

```
class RoundRobin : public smx_Task
{
public:
      RoundRobin( u8 pri ) : smx_Task( pri )  {  };

protected:
      virtual void Main(u32 par);
}

void RoundRobin::Main(u32 par)
{
      u8  pri = Priority();

      do
      {
            // perform operation

      } while( Bump( pri ) );
}

RoundRobin  TaskR(PRI_NORM);
```

In this example, a round-robin task class is derived from smx_Task and TaskR is defined using it. TaskR performs an operation, then bumps itself to the end of its priority level in the ready queue. This effectively creates a round-robin scheduling scheme.

BOOLEAN **smx_Task::Hook**()

Hooks the smx_Task::HookEntry() and smx_Task::HookExit() methods to this task. These must be defined for this task class since they are virtual no-ops in smx_Task..

smx functions called smx_TaskHook()

Returns none

Errors SMXE_OBJ_NOT_CREATED, SMXE_OP_NOT_ALLOWED

Example

```
class PreemptorC : public smx_Task
{
      PreemptorC(u8 pri, u32 stksz, u32 flags, cons char *name) : smx_Task(pri, stksz,
                  flags, name) {};
      virtual void HookEntry();
      virtual void HookExit();
      virtual void Main( u32 par );
}

PreemptorC::Main(u32 par)
{
      Hook();

      while(1)
      {
            // do main task function
      }
}

PreemptorC::HookEntry()
{
      /* smx_Task entry code goes here */
}

PreemptorC::HookExit()
{
      /* smx_Task exit code goes here */
}

Preemptor PreTask(2, 200, NO_FLAGS, "PreTask");
```

In this example, the PreemptorC class has been derived from the smx_Task class. PreTask is constructed from the PreemptorC class. When it first starts, it hooks the Preemptor HookExit() and HookEntry() methods. Thereafter, the scheduler will call these functions whenever PreTask is suspended or resumed, respectively.

virtual void **smx_Task::HookEntry**()

Entry routine called by the scheduler each time it resumes a hooked task. This is an empty stub, which must be overridden with an actual entry routine when a task class is derived from smx_Task().

smx functions called none

Returns none

See Also smx_Task::HookExit() and smx_Task::Hook()

Example See example for smx_Task::Hook().

virtual void **smx_Task::HookExit**()

Exit routine called by the scheduler each time it suspends a hooked task. This is an empty stub, which must be overridden with an actual exit routine when a task class is derived from smx_Task().

smx functions called none

Returns none

See Also smx_Task::HookEntry() and smx_Task::Hook()

Example See example for smx_Task::Hook().

void * **smx_Task::Locate**( )

Locates the queue which this task is in and returns a pointer to it.

smx functions called smx_TaskBump()

Returns

handle	pointer to queue this task is in
0	task is not in a queue or error

Errors SMX_OBJ_NOT_CREATED

Example

```
ReportC :: Main()
{
    void *qp = PreTask::Locate();
    switch (smx_SysWhatIs(qp))
    {
        case SMX_CB_NULL:
            ConIOManager.Print("MyTask is not in a queue");
            break;
        case SMX_CB_RQ:
            ConIOManager.Print("MyTask is in the ready queue");
            break;
        ...
    }
}
```

This example shows locating PreTask, defined above, then using the smx_SysWhatIs() service to determine what kind of queue MyTask is in and report it.

BOOLEAN **smx_Task::Lock**()

This method locks this task to prevent it from being preempted. This method is protected so that the current task can only be locked by itself.

smx functions called smx_TaskLock()

Returns TRUE ct locked.
FALSE ct is locked, but lock counter was not incremented.

Errors SMXE_OBJ_NOT_CREATED

Example See example for smx_Task::UnLock() method.

BOOLEAN **smx_Task::LockClear**()

Clears the lock nesting counter for this task thus permitting it to be preempted. This method is protected so that the current task can only be unlocked by itself.

smx functions called smx_TaskLockClear()

Returns TRUE ct is unlocked and its lock nesting counter is cleared.
FALSE ct is unlocked, but lock nesting counter was not 1.

Errors SMXE_OBJ_NOT_CREATED

Example

```
class HourTask : public smx_Task
{
public:
    u32 hour;
    virtual void Main( u32 )
    {
        Lock();
        hour++;
        LockClear();
    }
};
```

In this example, other tasks are blocked from accessing hour while it is being updated. Using lock clear to unlock assures that the task will be unlocked even if there were more locks than unlocks.

virtual void **smx_Task::Main**(u32 par)

This is an empty stub, which must be overridden when a task class is derived from smx_Task. Note that Main is a pure virtual method, which means that you will get a compiler error if you attempt to derive your own class from smx_Task without overriding Main().

smx functions called none

Returns none

Example See the example for smx_Task::Hook().

u32 **smx_Task::Peek**(SMX_PK_PARM par)

Returns the value of the specified parameter for this task. See smx_TaskPeek() in the smx Reference Manual for a list of semaphore parameters.

smx functions called smx_TaskPeek()

Returns

value	value for par.
0	value unless error.

Errors SMX_OBJ_NOT_CREATED

Example

```
u32 priority = TaskA.Peek(SMX_PK_PRI);
```

Returns the priority of TaskA.

u32 **smx_Task::RelAllBlocks**()

Releases all blocks owned by this task and returns the number released. May be useful if this task is being stopped or restarted. Note: called automatically if this task is being deleted.

smx functions called smx_BlockRelAll()

Returns u32 Number of blocks released

Errors SMXE_OBJ_NOT_CREATED

Example

```
u32 num = TaskA.RelAllBlocks();
```

u32 **smx_Task::RelAllMsgs**()

Releases all messages owned by this task. and returns the number released. May be useful if this task is being stopped or restarted. Note: called automatically if this task is being deleted.

smx functions called smx_MsgRelAll()

Returns u32 number of messages released

Errors SMXE_OBJ_NOT_CREATED

Example u32 num = TaskA.RelAllMsgs();

BOOLEAN **smx_Task::Resume**()

Removes this task from whatever queue it may be in and puts it into the ready queue at the end of its priority level. Does the equivalent of PipeResume() if this task is waiting at a pipe. Otherwise, does the equivalent of a timeout, unless this is the current task. If not the current task and it is now the top task and the current task is not locked, resumes it, if it was suspended or restarts it, if it was stopped. If the current task resumes itself, it is bumped to the end of its rq level and unlocked. If it is still the top task, it will continue, but it will be unlocked.

smx functions called smx_TaskResume()

Returns TRUE task resumed
FALSE error

Errors SMXE_OBJ_NOT_CREATED

Example

```
class MyTaskC : public smx_Task
{
    virtual void Main();
} MyTask

MyTaskC::Main()
{
    do {
        Lock();
        // main task function
    } while(MyTask->Resume());
}
```

This task resumes itself in order to implement round-robin scheduling. After doing its main function, it is requeued at the end of its priority level in rq, thus allowing the next equal-priority task in rq to run. Note that when this task resumes itself, it is unlocked, even if it continues immediately. Hence, if the main function is supposed to be locked, then Lock() must be called.

BOOLEAN **smx_Task::Sleep**(u32 stime)

Puts this task to sleep until the specified system time, stime. If stime < smx_stime the operation is aborted and returns FALSE. This method is protected so that only the current task can put itself to sleep.

smx functions called smx_TaskSleep()

Returns TRUE ct has been delayed
FALSE no delay due to invalid time

Errors SMXE_OBJ_NOT_CREATED

Example

```
class Sleeper : public smx_Task
{
    Sleeper( ) : smx_Task( (u8)PRI_NORM )  {  };
    void Main(u32);
} SleepT;

Sleeper::Main(u32)
{
    while (1)
    {
        // do main function
        Sleep(StimeGet() + 5 );
    }
}

AppInit()
{
    ...
    SleepT.Start();
    ...
}
```

In this example, the Sleeper class is derived from smx_Task class with its own main method. SleepT is an instance of this class, which is started by AppInit(). SleepT performs its main function, then sleeps for 5 seconds, wakes up, and repeats

void **smx_Task::SleepStop**(u32 stime)

Same as Sleep() method, except that the current task is stopped and then restarted instead of resumed.

smx functions called smx_TaskSleepStop()

Returns none

par TRUE current task has been delayed
FALSE no delay due to invalid time

Example
```
class Sleeper : public smx_Task
{
    Sleeper() : smx_Task((u8) PRI_NORM ) { };
    virtual void Main(u32);
} SleepT;

Sleeper::Main(u32)
{
    // do main function
    SleepStop(StimeGet() + 5 );
}

AppInit()
{
    ...
    SleepT.Start();
    ...
}
```

This example does the same thing as the previous example, except that SleepT stops after each iteration, then restarts when awakened. (Note that there is no while loop.) The advantage of this is that it releases its stack so other tasks can use it.

BOOLEAN **smx_Task::Set** (SMX_ST_PARM par, u32 val)

Provides task control.

smx functions called smx_TaskSet()

Returns TRUE success
FALSE error

Errors SMXE_OBJ_NOT_CREATED

Example
```
Atask::Main()
{
    ...
    Set(SMX_ST_STK_CK, OFF);
    // call function which changes stacks
    Set (SMX_ST_STK_CK, ON);
    ...
}
```

BOOLEAN **smx_Task::Start**()
BOOLEAN **smx_Task::Start**(u32 par)

Starts this task with or without a parameter by putting it into the ready queue at the end of its priority level. If this task is not currently stopped it will be stopped, then restarted. Either way, it will start from the beginning of its main function when it is dispatched. Also, it is given a stack from the smx stack pool.

smx functions called smx_TaskStart() or smx_TaskStartPar()

Returns TRUE task started
FALSE task not started due to error.

Errors SMXE_OBJ_NOT_CREATED

Example See smx_Task example.

BOOLEAN **smx_Task::Stop**(u32 tmo = INF)

Removes this task from any queue it may be in, stops it, and sets its timeout to tmo, which if not INF will result in this task restarting after tmo ticks. A task can stop itself, even if locked. A task becomes *dormant* if tmo = INF. In this state it will run again only if it is restarted by another task.

smx functions called smx_TaskStop()

Returns TRUE task stopped (not possible if task is stopping itself)
FALSE task not stopped due to error.

par/rv not affected

Errors SMXE_OBJ_NOT_CREATED

Example

```
class MyTaskC : public smx_Task
{
      MyTaskC( u8 pri ) : smx_Task( pri )  {};
      void Main( u32 par );
} MyTask( PRI_NORM );

MyTaskC::Main( u32 par )
{
      // do main function
      Stop(10);
}
```

In this example MyTask performs its main function then stops for 10 ticks, after which it is restarted and repeats. While stopped, MyTask releases its stack for use by other tasks. MyTask can be stopped by another task with: MyTask.Stop(tmo);

BOOLEAN **smx_Task::Suspend**(u32 tmo = INF)

Removes this task from any queue it may be in, suspends it, and sets its timeout to tmo, which, if not INF, will result in this task resuming after tmo ticks. This task can suspend itself, even if locked, in which case, it will resume after tmo ticks and it will be unlocked. A task can suspend another task for tmo ticks. See smx_TaskSuspend() in the smx Reference Manual for details.

smx functions called smx_TaskSuspend()

Returns TRUE task suspended
FALSE task not suspended due to error.

Errors SMXE_OBJ_NOT_CREATED

Example

```
class MyTaskC : public smx_Task
{
      MyTaskC( u8 pri ) : smx_Task( pri )   {};
      void Main(u32 par);
      BOOLEAN WorkToDo;
} MyTask( PRI_NORM );

MyTaskC::Main()
{
      while(1)
      {
            // perform main function
            Suspend(10);
      }
}
```

This example does the same thing as the previous Stop() example, except that MyTask is suspended rather than stopped. As a consequence, it keeps its stack. This would be useful if its stack contained information necessary to perform its main function.

BOOLEAN **smx_Task::UnHook**(void)

Unhooks the entry and exit routines from this task.

smx functions called smx_TaskUnhook()

Returns TRUE task unhooked
FALSE task not unhooked due to error.

Errors SMXE_OBJ_NOT_CREATED

Example See example for smx_Task::Hook()

BOOLEAN **smx_Task::UnLock**()

Decrements the lock counter. If it becomes 0, unlocks this task and tests for preemption. This method is protected so that current task can only be unlocked by itself.

smx functions called smx_TaskUnlock()

Returns TRUE task unlocked.
FALSE ct was already unlocked.

Example See example for smx_Task::Lock().

BOOLEAN **smx_Task::UnLockQuick**()

Decrements lock counter. If it becomes 0, unlocks this task but does not test for preemption. This method is protected so that current task can only be unlocked by itself.

smx functions called smx_TaskUnlockQuick()

Returns TRUE ct unlocked.
FALSE ct was already unlocked.

Example See example for Lock()

smx_Timer Class

file xtmr.cpp

#include xtmr.hpp

derived from smx_Object class

The smx_Timer class provides an interface to smx timers, which provide one-shot, cyclic, and pulse timers. An smx++ timer is initially created non-holding. When it is started, an smx timer is created, started, and linked to the smx++ timer. If an smx one-shot timer times out or is stopped, the smx++ timer becomes non-holding again.

Protected Data Members.

u32 **ObjectFlags**;
TMCB_PTR **TimerCBP**;

Constructors and Destructor

smx_Timer();
smx_Timer(smx_Timer *tmra, const char *name = NULL);
virtual **~smx_Timer**();

Methods

u32 **Peek**(SMX_PK_PARM par);
BOOLEAN **Reset**(u32 & tml);
BOOLEAN **SetLSR**(SMX_TMR_OPT opt, u32 par = 0, LSR_PTR lsr = NO_CHG);
BOOLEAN **SetPulse**(u32 period, u32 width);
BOOLEAN **Start**(u32 delay, u32 period, LSR_PTR lsr, const char *name = NULL);
BOOLEAN **StartAbs**(u32 time, u32 period, LSR_PTR lsr, const char *name = NULL);
BOOLEAN **Stop(** u32 & tml);
virtual SPP_CLTYPE **WhatIs**(void); // returns SPP_CL_TIMER

smx_Timer::smx_Timer()

Constructs a non-holding timer, which must then be started.

smx functions called none

Returns none

Example smx_Timer *tmrap = new smx_Timer();

Constructs a non-holding timer, tmra .

smx_Timer::smx_Timer(smx_Timer *tmra, const char *name = NULL))

Constructs a duplicate of tmra. If tmra is holding, its smx timer is duplicated and the new smx timer is enqueued after it. This timer will time out with tmra. If tmra is non-holding, tmrb will be also. See smx_TimerDup() for details concerning duplicate timers.

smx functions called smx_TimerDup()

Returns none

Example smx_Timer tmrb(tmrap, "tmrb");

tmrb is constructed from tmra. If tmra is non-holding, so will be tmrb and the name "tmrb" is not used.

smx_Timer::~smx_Timer( )

If this timer is holding, the underlying smx timer is stopped and deleted. Then this timer is destroyed.

smx functions called smx_TimerStop()

Returns none

Example delete tmrap;

Deletes the non-holding tmra constructed above.

u32 **smx_Timer::Peek**(SMX_PK_PARM par)

Returns the value of the specified parameter for this timer. See smx_TimerPeek() in the smx Reference Manual for a list of timer parameters..

smx functions called smx_TimerPeek()

Returns

value	value of par
0	value unless error

Example

```
u32 time_left

time_left = tmrc::Peek(SMX_PK_TIME_LEFT);
```

This will return the time left for tmrc, assuming that it is holding. If not holding, 0 will be returned, indicating that the timer has timed out, unless it was never started.

BOOLEAN **smx_Timer::Reset**(u32 & tml)

If this timer is holding, stops it and loads the time left of its current delay into tml. Then restarts this timer with its current delay and returns TRUE. If not holding loads 0 into tml and returns FALSE. The current delay is the length of the current interval, in ticks. Reset() is normally used with one-shot timers that put time limits on expected events. If the event occurs within the time limit, the timer is reset. See smx_TimerReset() in the smx Reference Manual for a more information.

smx functions called smx_TimerReset()

Returns

TRUE	timer reset
FALSE	timer not reset due to not holding or error

Example

```
u32 tml;

if (!tmrc.Reset(tml) && !SMX_ERR)
    // timer has expired
```

In this example, tmrc is reset and its time left is stored in tml. If Reset() fails and no smx error has occurred, then tmrc has expired (assuming it was started). This may indicate a problem or that its delay needs to be increased.

BOOLEAN **smx_Timer::SetLSR**(LSR_PTR lsr, SMX_TMR_OPT opt, u32 par)

If this timer is holding, changes its LSR, LSR option, and LSR parameter. The possible options are:

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

The parameter is the value returns for the SMX_TMR_PAR option. When a timer is started, the LSR option defaults to SMX_TMR_PAR and the LSR parameter defaults to 0.

smx functions called smx_TimerSetLSR()

Returns

TRUE	timer changed
FALSE	timer not changed due to not holding or error

Example

```
tmrc.SetLSR(SMX_TMR_STATE);
```

In this example, the LSR option of tmrc is changed to return the pulse state, which is more useful for the LSR, if tmrc is a pulse timer.

BOOLEAN **smx_Timer::SetPulse**(u32 period, u32 width)

If this timer is holding, changes its period and pulse width. Converts this timer from a cyclic timer to a pulse timer when first used with width > 0. Thereafter, simultaneously changing period and width allows various modulation schemes.

smx functions called smx_TimerSetPulse()

Returns
TRUE timer changed
FALSE timer not changed due to not holding or error

Example

```
tmrc.SetPulse(100, 10);
```

In this example, tmrc is changed to a pulse timer with period 100 ticks and width 10 ticks. If being used for pulse width modulation (PWM), pulse width could be increased a little with:

```
tmrc.SetPulse(100, 11);
```

This change will not task effect until the current period has completed. This assures smooth operation with no spikes.

BOOLEAN **smx_Timer::Start**(u32 delay, u32 period, LSR_PTR lsr, const char *name = NULL)

Makes a non-holding timer into a holding timer by creating an smx timer with the above parameters enqueueing it in the timer queue, and linking to this timer. If this timer is already holding, restarts the smx timer with the above parameters. When a timer is started, the LSR option defaults to SMX_TMR_PAR and the LSR parameter defaults to 0.

smx functions called smx_TimerStart()

Returns
TRUE timer is holding and has been started or restarted
FALSE timer not changed due to error

Example

```
void lsrA(u32 state);
smx_Timer  tmrc();

tmrc.Start(10, 100, lsrC, "tmrc");
tmrc.SetLSR(SMX_TMR_STATE);
tmrc.Pulse(100, 20);

void lsrC(32 state)
{
    if (state == ON)
        // turn on output 1
    else
        // turn off output 1
}
```

In this example, tmrc is constructed as an non-holding timer. It then is started as a cyclic timer with a delay of 10, a period of 100, and lsrC as its LSR. Then its LSR option is change to the timer state and it is changed to a pulse timer with period 100 and pulse width 20. When lsrC runs, its state parameter indicates if the pulse is ON or OFF. In this case, lsrC simply toggles output 1, accordingly.

BOOLEAN **smx_Timer::StartAbs**(u32 time, u32 period, LSR_PTR lsr,
const char *name = NULL)

Operates exactly like Start(), except that time is the time from system start (i.e. smx_etime == 0).

smx functions called smx_TimerStart()

Returns

TRUE	timer is holding and has been started or restarted
FALSE	timer not changed due to error

Example

```
void lsrA(u32 state);
smx_Timer  tmrc();

tmrc.StartAbs(100, 100, lsrC, "tmrc");
tmrc.SetLSR(SMX_TMR_STATE);
tmrc.Pulse(100, 20);

void lsrC(32 state)
{
      if (state == ON)
            // turn on output 1
      else
            // turn off output 1
}
```

Operates like the previous example, except that tmrc does not start until etime == 100.

BOOLEAN **smx_Timer::Stop**(u32 & tml)

Allows stopping this timer without destroying it. If this timer is holding, stops the smx timer, loads the time left of its current delay into tml, and returns TRUE. If not holding, loads 0 into tml and returns TRUE. If this timer is not holding, then it has either stopped or never been started. smx++ cannot distinguish between these alternatives.

smx functions called smx_TimerReset()

Returns

TRUE	timer stopped or was already stopped
FALSE	timer not reset due an error

Example

```
u32 tml;

tmrc.Stop(tml);
```

In this example, tmrc is stopped and its time left is stored in tml.

Appendix A: smx++ Memory Management

There are three ways that memory space is allocated for C++ objects. These are discussed below.

Local Objects

C++ objects need not be created by explicit calls to the new operator. It is possible to declare a C++ object as a local variable, in which case it is referred to as a *local object* or *local instance*. In this case, the compiler allocates the memory for the object from the current stack and automatically calls the appropriate constructor when the object comes into scope. The destructor is called when the object leaves scope. A Local Object can be convenient for objects that are only needed within a block of code. For example:

```
void MsgFill(void)
{
    smx_Msg msgA(80);
    ...
}
```

This creates a message of 80 bytes for use within MsgFill(). When it returns, msg is destroyed. Before then, it probably will have been sent to an exchange and thus be a non-holding message when destroyed.

Space for local objects is allocated from the current stack, when the function in which they are defined, is called. This is very fast. Space is released when the function returns, which is also very fast. Inside of the function, local objects are easy to use and have minimal overhead. Hence, they seem very attractive.

The downside for using local objects in a multitasking system, is that space for a class must be provided in the stack of every task using its objects. Since only one task can run at a time, this results in wasted storage. If deep nesting of many functions using many objects occurs, task stacks can become quite large. The net result could be forcing task stacks out of local memory into much slower external memory. If this happens, performance will take a serious hit.

This problem can be ameliorated by using one-shot smx tasks, which share stacks from a stack pool. A shared stack is given to a task only when it runs. In a typical system, higher-level tasks may tend to run in sequence and not have tight deadlines. Such tasks may be idle most of the time and may not need to carry information over from one run to the next. Hence, it is practical to share a few stacks among many such tasks. As a result, there can many fewer stacks and space for objects in them becomes less of a problem. For further discussion of one-shot tasks, see the smx User’s Guide.

Global Objects

A global object is an instance of a class declared outside of any block, such as:

```
smx_Msg msgB(80);
```

Unlike msgA, msgB is not defined within any scope. Hence, it is a global object. Space for a global object, is statically allocated from memory, by the compiler. But since it is an object, its constructor must be called to initialize it before it can be used. To do this the compiler inserts calls to the constructors of all global objects into a table of constructor pointers. The startup code steps through this table and executes all constructors found therein. This is done before reaching main().

Global objects must also be destroyed at the end of the application by calling their destructors. The destructors are similarly placed into a table and exit code steps through this table and executes all destructors found therein.

There is no guarantee of order of execution of either constructors or destructors. If this is important for your application do not use global objects. Instead, use global pointers and create your objects using the global new operator, as discussed next.

Global new and delete Operators

The new and delete global operators are used to dynamically allocate and free memory for C++ objects, as follows:

```
char * bp = new char[1000];
...
delete[] bp;
```

In this example, a buffer of 1000 bytes is allocated from the heap. bp is a static pointer, which points to it. When no longer needed, the buffer is released back to the heap by calling delete[]. However, the bp pointer survives the delete operation.

smx replaces the standard C++ global new(), new()[], delete(), and delete()[] operators with operators that make smx heap calls instead of compiler heap calls. smx heap calls are safer for multitasking environments and offer other advantages over compiler heap calls. Replacement is done in heap.c and xapi.h.

Note: If the object is of a class derived from smx_Object, then the latter's new and delete operators will be used instead of the global operators. :: can be used to force use of global operators, for example:

```
char * bp = ::new char[1000];
...
::delete[] bp;
```

The biggest advantage of using new and delete is that memory is dynamically allocated as objects are needed, then released as they are deleted. This can result in significant memory savings vs. using the local or global objects, discussed above. On the other hand, static objects do not have allocation and deallocation overhead. So the best choice depends upon the size of an object and how frequently it is constructed and destructed.

A significant problem with using new is that delete is not automatically called when an object created with new goes out of scope. delete must be called for the object, else a memory leak will occur.

Operator and Function Descriptions

void operator **delete**(void * block)

This overrides the global delete operator used to free space used by C++ objects. It releases the memory block to the smx heap. This should only be used on objects allocated via the global operator new().

smx functions called smx_HeapFree()

void operator **delete**[](void * block)

This version of the global delete operator is used to free memory allocated via the global operator new[]. The implementation is identical to delete().

smx functions called smx_HeapFree()

void * operator **new**(size_t s)

This overrides the global new operator used to allocate memory for C++ objects. It allocates *s* bytes from the smx heap.

smx functions called smx_HeapMalloc()

See Also newx()

Example ExitxTp = ::new smx_Task(smx_ExitTask);

Notice that the global version of new must be called out specifically (i.e. *::new()* instead of just *new()*) if we want to use it to create an smx++ object.

void * operator **new**[](size_t s)

This operator is to be used for allocating an array of objects.

smx functions called smx_HeapMalloc()

See Also newx()

Example char * Buffer = new char[26];

Creates an array of 26 characters.

void * **newx**(size_t s)

This internal routine is called by the new operator, above. It calls smx_HeapMalloc() to allocate *s* bytes from the smx heap. If the allocation succeeds, the address of the allocated block is returned. If it fails, a check is made to see if a new_handler has been installed. If not, NULL is returned. If so, it is called allocate *s* bytes. If the new_handler returns a non-zero value, the allocation is retried, else it returns NULL. Initially there is no new_handler installed.

smx functions called smx_HeapMalloc()

See Also new()

inline PNHX **set_new_handler**(PNHX new_handler)

set_new_handler sets the function to be called if the global new operator fails to allocate the requested memory. The argument is the address of a user defined function which is to handle such cases where memory can not be allocated from the heap. It must have the following format:

```
typedef int (* PNHX)( size_t );
```

The *size_t* argument to the new_handler function is the size of the object being created. If the new_handler can make memory available in the heap, then it may return a non-NULL value to cause the allocation to be retried. Otherwise, it should return NULL, which will cause new() to fail and the object will not be created.

Note that

```
BOOLEAN  smx_HeapRecover(u32 sz, u32 num)
```

or

```
BOOLEAN smx_HeapExtend(u32 xsz, u8* xp)
```

could be used to implement this.

smx functions called none

See Also newx()

Example

```
int new_handle(size_t sz)
{
    return ((int)smx_HeapRecover((u32)sz, INF));
}
```

This will run through the entire heap attempting to find adjacent free blocks to form a big-enough chunk. Higher priority tasks could be blocked from running for a long time. Hence, it is probably better to call HeapRecover() repetitively with num equal to a smaller value. However, this may still block the current task and tasks of equal or lower priority for too long. smx_HeapExtend() might be the better solution if more memory is available.

Appendix B: smx++ Error List Additions

This appendix describes the new errors that were added for smx++. They were added to clarify the dynamic nature of this new version.

SMXE_HOLDING An smx_Msg object is already holding a message and cannot receive another one.

SMXE_NOT_HOLDING An smx_Msg object is not holding a message, so the Send, Put, or other operation cannot be performed.

SMXE_OBJ_IN_USE This error occurs when the destructor is called on an object which is still in use, an smx_Pool whose blocks are still being used for example.

SMXE_OBJ_NOT_CREATED Indicates that an object was not created successfully because it could not get an smx control block. If the control block pointer is NULL then this error is reported.

Appendix C: smx++ Limitations

smx Functions Not Implemented

The following is a list of smx API functions that were not implemented in smx++.

smx_TaskLocate()

For the locate functions to be meaningful in smx++, they would have to return the class instance that the message or task was in. We can no longer use the control block since the user does not have access to it in smx++.

Appendix D: Additional References

1. smx++ datasheet: Overview of smx++.
2. smx Users Guide: Discusses the workings of smx.
3. smx Reference Manual: Details of smx API services and glossary of all terms.
4. Effective C++ Third Edition by Scott Meyers: A great book on C++ programming.